VAX/VMS Software
Language and Tools Handbook

VAX/VMS Software
Language and Tools Handbook

digital

Contents

Chapter 2 ▪ VAX Program Development Productivity Tools

Chapter 3 ▪ VAX Programming Languages

Chapter 5 ▪ VMS Program Development Utilities

Chapter 6 ▪ VAX Program Migration and Cross-Development Tools

Chapter 7 ▪ Language and Tool Integration in the VAX/VMS Environment

List of Figures

List of Tables

Preface

The *VAX/VMS Languages and Tools Software Handbook* is a comprehensive reference guide to the latest software development products available from Digital. It is part of a three-volume set titled, *VAX/VMS Software.* The other handbooks in this set are, *The VMS System Software Handbook* and *The VAX Information Management Handbook.*

Digital has developed many products to meet the varied needs of today's program development shop. These include industry-standard, high-level language compilers, program preparation and development tools, and even many tools that allow programmers familiar with other operating systems to take advantage of the power and flexibility of the VMS software development environment.

Handbook Organization

- Chapter 1 — "The VAX/VMS Software Development Environment" gives you an introduction to the family of VAX/VMS program development software products and the single integrated environment they are part of.

  If you are not familiar with these software products and their relationship to each other and to the VMS operating system, this chapter will give you the overview you need before proceeding to the specifics of VAX/VMS program development software products.

- Chapter 2 — "VAX Program Development Productivity Tools." This chapter describes the many VAX productivity tools that can be used in conjunction with VAX languages. These tools streamline the program development environment by giving programmers control over the many tedious and time-consuming aspects of "building" software systems.

  Each tool is covered in its own section of the chapter and includes a general description of the tool's features and benefits.

- Chapter 3 — "VAX Programming Languages". This chapter describes the most widely used VAX Languages. It is organized alphabetically by language name and includes the features, benefits, and primary applications of each language. Detailed information on commands and compilers, and a sample program listing is also included.

- Chapter 4 — "VMS Program Development Services". The foundation on which VAX productivity tools and languages operate is the VMS operating system. Chapter 4 describes four services provided by the operating system and used in program development. These include, the Digital Command Language (DCL), Digital Record Management Services (RMS), the VAX Runtime Library (RTL), and VMS System Services.

- Chapter 5 — "VMS Program Development Utilities." This chapter gives you an introduction to VMS's program development utilities. Included are descriptions of VAXTPU and EDT text editors, the VAX Symbolic Debugger, Linker, Librarian, and many other sophisticated utility programs that are an integral part of the VAX/VMS program development environment.

- Chapter 6 — "VAX Program Migration and Cross-Development Tools." Three optional VAX/VMS software products — VAXELN, VAX-11 RSX, and MicroPower/Pascal-VMS — make it possible for you to use your VAX system as a host program development system to develop applications that can then be run on other Digital systems. This chapter introduces you to both of these products.

- Chapter 7 — "Language and Tool Integration in the VAX/VMS Software Development Environment." This chapter gives you a brief overview of the program development life cycle as we define it at Digital and shows you how our software products interact to make each phase of the development cycle as productive and cost effective as possible.

Scope of the VAX/VMS Handbook Set

It is not the intent of this handbook set to describe all VAX/VMS software products. Products covered in these three volumes pertain to the subject of the specific handbook they appear in. For example, the VAX *Languages and Tools Handbook* describes VAX/VMS software products used for software development. Many product not discussed in these three volumes are covered in other handbooks or related documentation.

▪ Related Publications

The following publications can provide you with additional information on VAX/VMS software products discussed in this Handbook Set and can be obtained through your local Digital Sales Office or Sales Representative. Or, in the United States, for more information, an online demonstration of many of the products discussed in this handbook, or to purchase a product, call:

Digital's Electronic Store (1-800-332-3366) with a VT100 compatible terminal and a 1200 baud modem and follow the directions that appear on your screen.

- You can find specific hardware/software support guidelines for VAX/VMS software products by referencing the Software Product Description (SPD) of a particular product. Information in the Alphabetical Index (SPD 00.01.11), The VMS Operating System SPD (25.01.22), and The VAX/VMS Optional Software Cross-Reference Table SPD (25.99.37) is particularly useful in determining the type of VMS operating system support and the SPD number of a product.

- *VAX Software Source Book* Volume 1, Application Software; Volume 2, System Software

- *The Digital Dictionary* provides generic and Digital-specific definitions for the technical terminology found in this handbook set.

- *VAX/VMS Internals and Data Structures*. This book describes indepth the VMS operating system's executive and a number of its related subsystems.

- User and reference manuals can also be obtained for each of the products described in this handbook set. You can find out how to order these manuals for the software products running of your specific VAX processor in the *Peripherals and Supplies Group's Documentation Products Directory.*

Introduction to VAX/VMS Software

Computing resources, hardware and software, make it possible for an organization to effectively deal with a multiplicity of both day-to-day and long-range operating needs. Typically, these include data processing, program development, and information management requirements.

A grouping of computer resources creates a computing environment. To be effective, a computing environment must be made up of resources that are compatible with each other and designed to work together toward a common goal. Digital's VAX/VMS software products are designed to create such a computing environment — the VAX/VMS Productivity Environment.

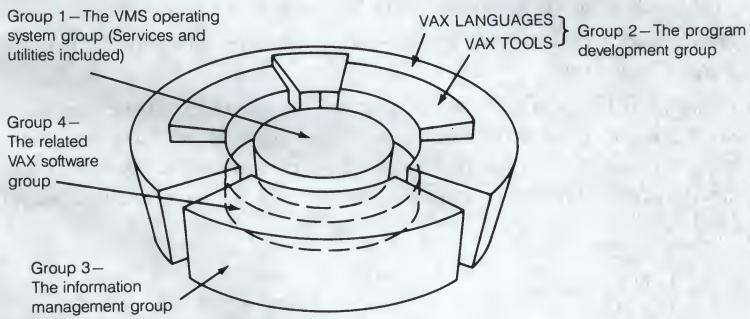
The VAX/VMS Productivity Environment

- THE VMS PRODUCTIVITY ENVIRONMENT'S FOUR SUBSETS

  Digital's VAX/VMS productivity environment is made up of many different software products. Even though all these products have been designed to operate as a single integrated environment, they are can be organized, by function, into four logical groups. Each group is a subset of the overall VAX/VMS productivity environment and provides VAX/VMS users with specific capabilities.

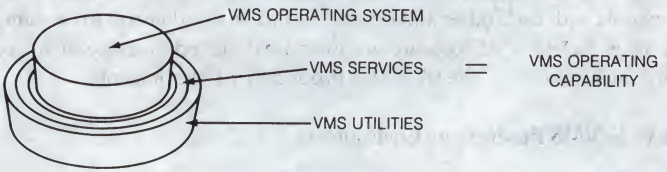
  These four groups of software products are the

  1. VMS operating system group (including services and utilities).
  2. VAX program development group.
  3. VAX information management group.
  4. Related VAX software group.



- THE VMS OPERATING SYSTEM GROUP (1)

When you buy a VAX/VMS system, in addition to the various hardware components, you receive a standard set of software programs essential for the basic operation of your system. This group of software is the VMS Operating System, a number of services provided by the operating system, and many utility programs used for system management and program development. This group serves as the foundation on which all VAX optional software products (those products in groups 2-4) and applications programs generated from those products operate.



The VMS Operating System, Services, and Utilities

The VMS Operating System gives you the flexibility and control you need to successfully distribute computing resources throughout your organization's computing environment. A few of the many unique features built into the VMS operating system give you the ability to cluster, network, and secure resources within your computing environment.

VMS services include the Digital Command Language, VAX Record Management Services, the VAX Runtime Library, and VMS System Services.

VMS Utilities are grouped into program development and system management utilities.

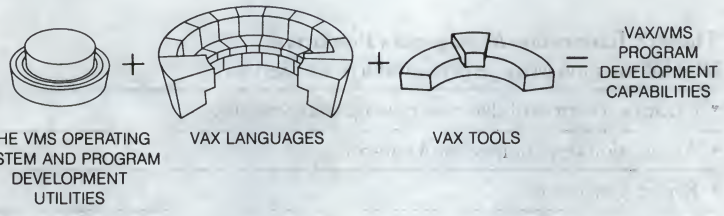
The MicroVMS operating system supports Digital's MicroVAX processors and is a full function, special packaging of the VMS operating system.

For more information on the VMS operating system group see the *VMS System Software Handbook.*

Digital also offers you many optional software products that can be purchased in addition to basic system software. These optional products have been designed to work in conjunction with VMS system software and perform a specific function. Software described in the next three groups are all optional products.

- THE VAX PROGRAM DEVELOPMENT GROUP (2)

This group of optional software products is made up of a rich set of VAX programming languages and VAX program development productivity tools. With this group of products (and many of the services and program development utilities provided with the VMS operating system), you can build software that runs on the full spectrum of VAX processors — MicroVAX/VAX Station to VAX 8600 VAXcluster systems.



The VAX/VMS Program Development Products

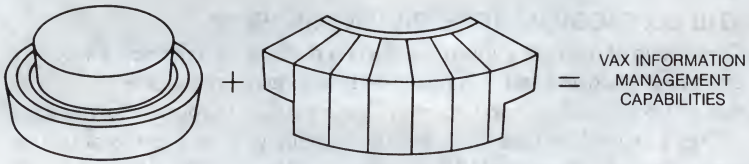
Included in the VMS program development software group are

- Over a dozen high-level (sixteen in all), industry-standard programming languages that are callable from each other — including VAX™ Ada$, VAX APL, VAX BASIC, VAX BLISS, VAX C, VAX COBOL, VAX DIBOL, VAX FORTRAN, VAX PASCAL, VAX PL/I, and VAX RPG II.
- Numerous program development productivity tools that simplify many of the tedious and time-consuming chores involved in large-scale and labor-intensive, small-scale software development.
- Tools that aid in building software systems efficiently and reliably.

- THE VAX INFORMATION MANAGEMENT SOFTWARE GROUP (3)

Digital's VAX Information Management Software group offers you a variety of integrated software products to meet the diverse information management needs of your organization. Products in this group run on the VMS operating system and can be used with products in groups 1,2, and 4. For example, the Common Data Dictionary (CDD) is often used extensively with products from the Program Development Group (2) — VAX languages and tools — when programmers build application programs.

*Ada is a registered trademark of the U.S. Department of Defense.



The VAX Information Management Products

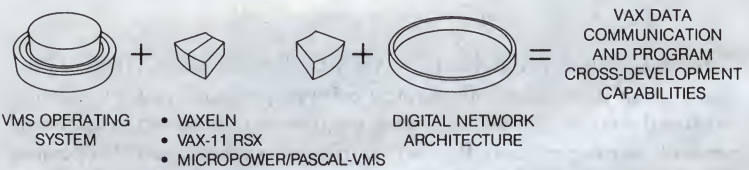
VAX Information Management products are used for

- Creating a variety of database management systems.
- Application development and control.
- Report generating.
- Easy-to-use querying.

A description of the VAX information management products and its associated software products can be found in *The VAX Information Management Handbook*.

- THE RELATED VAX SOFTWARE GROUP (4)

Related VAX software products enhance the capabilities found in the three groups of software products just discussed. These products are used to move and develop software products between DEC systems and make it possible for VAX systems to communicate with other DEC systems in a data communications network.



Chapter 1 ▪ The VAX/VMS Software Development Environment

▪ Introduction

To help you better manage your software assets, Digital offers you the VAX/VMS Software Development Environment. This highly integrated environment comprises software products designed to a common specification and embodying a common set of characteristics. These products run on a single operating system (VMS) and give you and other programmers a consistent environment for the design, implementation, testing, and support of software programs.

These products help you

- Organize and manage your software projects more economically.
- Build more reliable code.
- Code and debug programs easier.
- Understand your product and its performance.
- Manage changes to code and build your system efficiently.
- Control the evolution of *your* programming environment.

The integration of the VMS Operating System, VAX Languages and Tools, and Program Development Utilities is made possible by

- A common architecture to which all these products conform.
- Common runtime capabilities available to all applications that are created and run on the VMS operating system.

The VAX Common Language Environment

When we first designed the VAX/VMS Software Development Environment, it was necessary to create standards to control the design and implementation of products that would exist within that environment. These standards ensures all products, their future releases, and any new VAX product offerings are compatible with the existing products in the environment. This common design goal is the VAX Common Language Environment.

The VAX Calling Standard, the Guide to Modular Procedures, and the VAX Condition and Exception Handling Standard are the basis of the VAX Common Language Environment. The calling standard defines the mechanisms for passing arguments between program modules. The Guide to Modular Procedures provides a consistent set of software programming pragmatics, and the Condition and Exception Handling Standard defines the mechanisms that ensure consistency in error and exception handling routines, regardless of the mix of programming languages in use. For example, in no other program development system on the market today, can you write a program module in Basic that calls another module in written FORTRAN and then calls a math routine written in Pascal — all in one Ada program.

As an extension of the VAX Common Language Environment, our VAX tools have a number of common characteristics.

- They are all based on the VMS Operating System and the VAX Common Language Environment.
- They can be used with many VAX languages — either they are language neutral or they provide capabilities that allow you to tailor them to support the various languages they are aimed at.
- They can be used with many different designs or methodologies.
- They have been designed to help you in portions of the software development task that consume large amounts of your time.
- The tools are designed to be consistent in terms of user input and response to that input. They use the same standard command language, prompts, and error messages.
- The tools are based on a compatible set of data formats — the VAX data types, descriptors, and calling standard.
- Many of these tools can be extended and tailored to your unique requirements. This helps you adapt our set of products to your local needs, whether it is customs and standards or defaults that are particularly appropriate to your shop.

VMS Runtime Capabilities

A few important runtime capabilities of the VMS operating system are

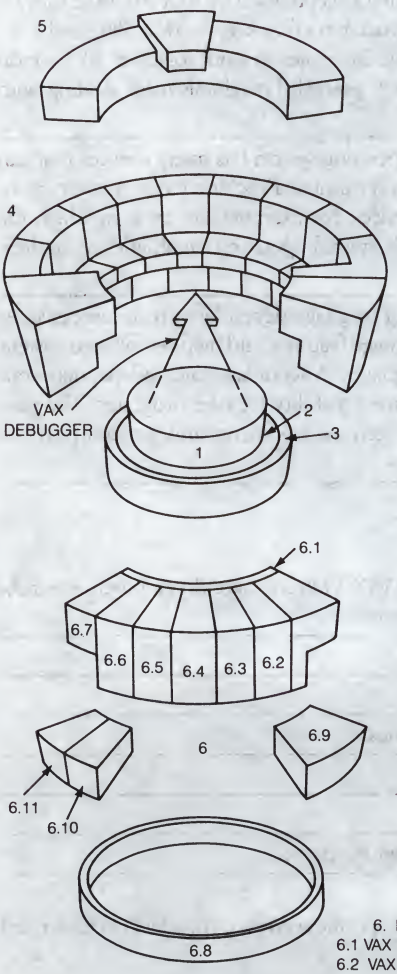
- A set of routines that manage record and file I/O for all languages the VAX Record Management Services (RMS).
- A common mechanism for handling exceptions. The VAX Runtime Library (RTL) provides a common set of procedures for exception handling and common-resource handling that enable languages to work together. VAX condition handling provides a low-level, powerful mechanism for dealing with exceptions in various languages.
- VMS system services. The VMS operating system has many services that can be used by an application program at runtime. Process control, memory management, and system security services, for example, are areas in which the VMS operating system can provide special operating capabilities to application programs at runtime.
- The VMS Runtime library. A set of language-dependent procedures ensures correct operation of complex language features, and helps enforce consistent operations on data across the languages. A set of language-independent routines establishes a common runtime capability for user programs. The runtime library has many math, screen-management, and general-purpose procedures.

Services and Products

The software facilities found in the VAX/VMS software development environment are divided into six basic categories.

- The VMS Operating System.
- VMS Services.
- VMS Program Development Utilities.
- VAX Languages.
- VAX Software Tools.
- Related VAX Program Development Products.

Each category is further divided into specific services or products as illustrated in Figure 1-1.



VMS CORE SERVICES

1. The VMS Operating System

2. VMS Services
   - The Digital Command Language (DCL)
   - The VMS Runtime Library (RTL)
   - VMS system services
   - The VAX Record Management Services (RMS)

3. The VMS Program Development Utilities
   - The VAX/VMS Symbolic Debugger
   - Sort/Merge
   - The VAX Linker
   - The VAX Librarian
   - VAXTPU text processing utility
   - DSR text-formatting utility
   - Mail

VAX OPTIONAL PROGRAM DEVELOPMENT PRODUCTS

4. VAX Languages
   - VAX Ada
   - VAX Apl
   - VAX BASIC
   - VAX BLISS
   - VAX C
   - VAX COBOL
   - VAX CORAL
   - VAX DIBOL
   - VAX DSM
   - VAX FORTRAN
   - VAX LISP
   - VAX PASCAL
   - VAX PL/I
   - VAX RPG II
   - Other VAX Languages

5. VAX Software Productivity Tools
   - VAX DEC/Code Management System (CMS)
   - VAX DEC/Module Management System (MMS)
   - VAX Language-Sensitive Editor (LSE)
   - VAX Performance and Coverage Analyzer (PCA)
   - VAX DEC/Shell
   - VAX DEC/Test Manager
   - VAX Graphical Kernel System (GKS)

6. Related Program Development Products

6.1 VAX CDD	6.7 VAX FMS
6.2 VAX DBMS	6.8 DECnet Communications Products
6.3 VAX Datatrieve	
6.4 VAX Rdb/VMS (ELN)	6.9 VAXELN
6.5 VAX ACMS	6.10 VAX-11 RSX
6.6 VAX TDMS	6.11 MicroPower/Pascal-VMS

Figure 1-1 ▪ VMS Services and Products used for Program Development

The VMS Operating System

VMS is a general-purpose operating system widely used for the simultaneous execution of timesharing, batch, or realtime application programs by many users.

Applications can be developed and run across the entire line of VAX processors, within limits of existing program size and available memory. User-mode applications developed with VMS will run on MicroVMS without modification.

The VMS Operating System gives you extensive online help. You can receive help by using the HELP command. The HELP facility provides you with information on the syntax used to invoke the languages, services, and utilities supported by the system. In many instances, HELP text also includes examples that use those commands. Besides being able to access HELP at the operating system level, many of the software products that run on VMS have their own HELP facilities that can be accessed while using those software products. For example, HELP can be requested from within VMS editors, the VMS Mail Utility, and some language development environments.

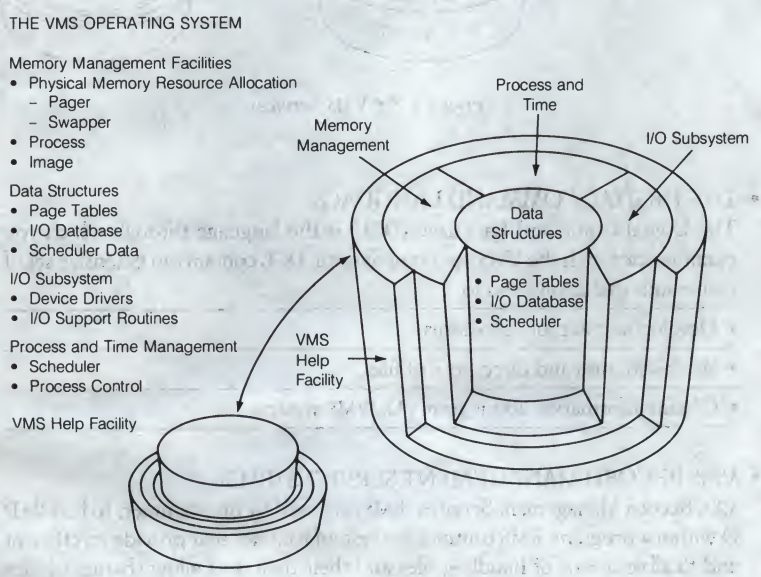


Figure 1-2 ▪ The VMS Operating System

For more information on the VMS operating system, see the *VMS System Software Handbook* and *VAX/VMS Internals and Data Structures**.

*Kenah and Bates, 1984, The Digital Press, Order no. EY-00014-DP

The VMS Services

Complementing the operating system are four layers of services essential for basic system operation and program development. These are called the VMS services and include

- The Digital Command Language (DCL).
- The VAX Runtime Library (RTL).
- The VAX Record Management Services (RMS).
- The VMS System Services.

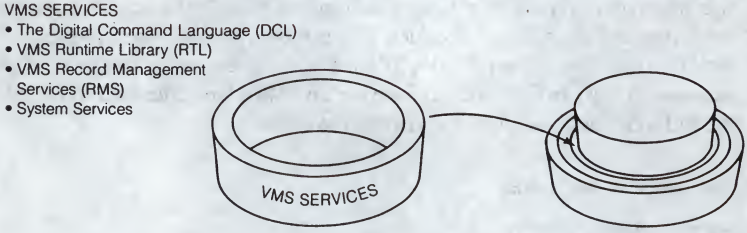


Figure 1-3 ▪ VMS Services

- THE DIGITAL COMMAND LANGUAGE

  The Digital Command Language (DCL) is the language through which you communicate with the VMS operating system. DCL contains an extensive set of commands that allows you to

  - Develop and execute programs.
  - Work with files and directories of files.
  - Obtain information about your VAX/VMS system.

- VMS RECORD MANAGEMENT SERVICES (RMS)

  VAX Record Management Services (RMS) are used by programmers to handle I/O within a program. RMS routines are system routines that provide an efficient and flexible means of handling files and their data, and allow sharing of files across languages.

RMS is the default I/O service for all VAX languages. VAX languages in the VAX Common Language Environment use identical representations for many data types. Therefore, files associated with a program written in one VAX language can be read and used by another program, even if it is written in a different language. VAX RMS can also be used with Digital's DECnet-VAX software to manipulate files across many VAX systems. (For more on using RMS with Digital Network Architecture products, see the *VMS System Software Handbook*, Chapter 6.)

The RMS Utilities and a File Definitions Language (FDL) complement RMS procedures by providing additional capabilities for creating and maintaining RMS files.

For more information on VAX RMS see Chapter 4 of this handbook.

- THE VMS RUNTIME LIBRARY (RTL)

The VMS Runtime Library's set of language-independent procedures reduces the time it takes you to design and implement an application program. These procedures can be called from any language in the VAX Common Language Environment.

Because application programs can call VMS general-purpose routines (data management, for example), you can skip designing many of those elements and concentrate on the central application.

The Runtime Library provides runtime support for VAX high-level languages. All procedures in the Runtime Library follow all standard call and condition handling conventions.

The common Runtime Library provides language-support procedures general string manipulation, I/O and I/O conversions, terminal-independent screen handling, and many more procedures.

For more information on the VAX RTL, see Chapter 4 of this handbook.

VMS Program Development Utilities

A few of the many VMS program development utilities include

- VAXTPU (text editor)
- EDT (text editor)
- DSR (text formatter)
- VAX/VMS Symbolic Debugger
- VMS Linker
- VMS Sort/Merge
- Other program development utilities

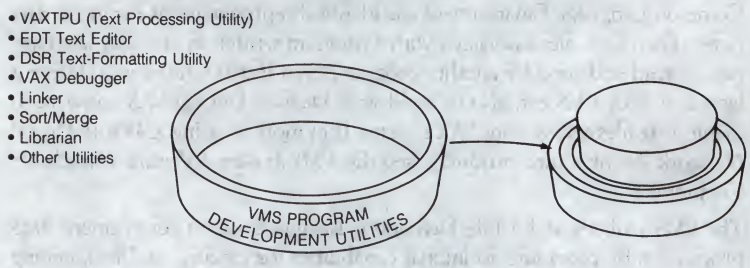


Figure 1-4 ▪ VMS Program Development Utilities

- VAXTPU (TEXT PROCESSING UTILITY)

  VAXTPU is a high-performance programmable text processing utility. VAXTPU is a tool designed to aid application and system programmers in the development of text processing interfaces. The utility includes a compiler, an interpreter, a high-level procedural language, and two editing interfaces written in VAXTPU. One interface emulates the EDT text editor while the other was built based on interactive human factors testing.

  You can tailor one of the existing VAXTPU editing interface to suit your editing style, or you can write your own editing interface with VAXTPU. You can use VAXTPU to design an intelligent editor for a specific environment.

  For more information on VAXTPU, see Chapter 5 of this handbook.

- THE EDT TEXT EDITOR

  EDT is Digital's easy-to-learn editor ideally suited for the novice or general-purpose user. In line mode, you can issue commands to EDT to change a single line of text or many lines. In keypad mode, EDT is a character-oriented editor and enables advanced users to edit any form of ASCII files — program source files, manuscripts, or correspondence. Because EDT is keypad oriented, it does not require an entire line of text to be replaced each time a character is changed. EDT also has an extensive HELP facility, which is available online during an editing session.

  For more information on the EDT Text Editor, see Chapter 5 of this handbook.

- THE DSR TEXT-FORMATTING UTILITY
  Digital's Standard Runoff (DSR) text-formatting utility extends the basic functions of VMS's text editors into a sophisticated text processing system, ideal for creating and maintaining the extensive documentation necessary to support any program development effort. With DSR's powerful command set, you can create documentation ranging from a simple form letter to a multichaptered manual.

  For more information on the DSR Text-formatting utility, see Chapter 5 of this handbook.

- THE VAX SYMBOLIC DEBUGGER UTILITY
  The VAX/VMS Symbolic Debugger utility is a powerful and flexible tool that aids you in locating errors in programs.

  The DEBUGGER

  - Is interactive. You can execute debugger commands from your terminal and see their effects immediately.
  - Is symbolic. You can refer to program locations by the symbols you used for them in your program.
  - Supports many languages. If your application is written in more than one language, you can change from one language to another in the course of a debugging session.
  - Permits a variety of data forms and types for entry and display.
  - Allows you to select and display your program's language statements.
  - Has a screen mode that provides multiple windows for screen-oriented debugging.
  - Has a debugger-defined keypad key definitions for many types of terminal's numeric keypad.
  - Provides online help.

See Chapter 5 of this handbook for more information on the VAX/VMS Symbolic Debugger.

Other VMS utilities used for program development include

- The Linker Utility (LINK).
- The SORT/MERGE Utilities.
- The LIBRARIAN Utility.
- The Command Definition Utility (CDU).
- The MESSAGE Utility.
- The DIFFERENCES Utility.
- Other VMS program development utilities.

See Chapter 5 of this handbook for more information on each of these and other VMS utilities. For a discussion of the VMS system management utilities, see the *VMS System Software Handbook*.

Optional VAX Program Development Products

- VAX LANGUAGES

At the center of the VAX/VMS software development environment is the family of VAX programming languages. Applications and system programmers have a diversified range of higher-level programming languages at their disposal. In addition to the assembly-level language, VAX MACRO, VAX programming languages range from Ada to RPG II. For an indepth coverage of each of the languages introduced below, turn to Chapter 3 of this handbook.

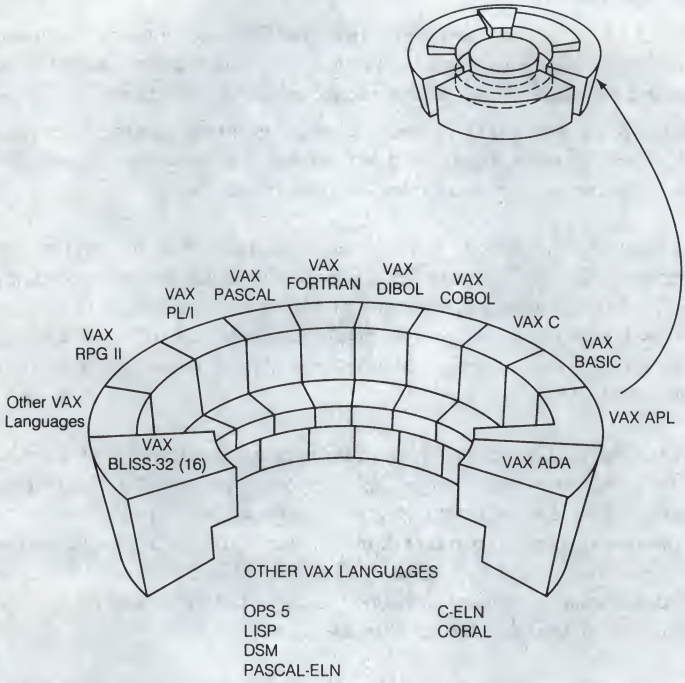


Figure 1-5 ▪ VAX High-Level Programming Languages

VAX Ada - Ada is a modern, higher-order programming language designed as a result of a competition sponsored by the United States Department of Defense. Although Ada has now become the single programming language for all mission-critical Department of Defense software, it is also well suited to many civilian applications, such as CAD/CAM, or process control. Ada is ideal for large applications that must be developed and maintained by many programmers.

VAX APL - VAX APL (A Programming Language) is a compact and versatile programming language that runs on the VAX series of computers. This language is especially suited to handle numeric and character data organized as lists and tables. It is used extensively in such areas as the manipulating of data, the designing of systems, and the computing of mathematical and scientific solutions.

VAX BASIC - The VAX BASIC product gives you the benefits of a highly interactive programming environment and a high-performance development language that is fully integrated with the VAX/VMS Software Development Environment.

The VAX BASIC language provides a structured language, powerful mathematical and string handling facilities, support for symbolic variable names/debugging, and full RMS indexed, sequential, and relative I/O operations.

VAX BASIC can be used as if it were either an interpreter or a compiler. A fast RUN command and support for direct execution of unnumbered statements (immediate mode) gives you the feel of an interpreter.

VAX Bliss-32 - VAX Bliss-32 is a high-level systems implementation language. The Bliss-32 language supports development of modular software according to structured programming concepts by providing an advanced set of language features. It provides access to most of the hardware features of the VAX systems to facilitate programming of time-critical and hardware-dependent applications.

VAX C - VAX C fully supports many of the language features of C, as described in *The C Programming Language**. VAX C provides program flow control constructs for logical and efficient program structuring, and a rich assortment of operators and common run-time routines (only those UNIX-specific routines that cannot be reasonably emulated under VAX/VMS are omitted.) VAX C even includes language extensions developed since the Kernighan and Ritchie book was published, including the structure assignment feature.

VAX COBOL - VAX COBOL is a high-performance implementation of COBOL. It is based on American National Standard Programming Language COBOL, X3.231974, the industry-wide accepted standard for COBOL. Most features planned for the next COBOL standard, based on the specifications in the Draft Proposed Revised X3.23 American National Standard Programming Language COBOL, are also included.

VAX COBOL also supports an embedded Data Manipulation Language (DML) interface to VAX DBMS, Digital's CODASYL compliant Database Management System. Also, it allows access to common record definitions stored in the VAX Common Data Dictionary. VAX COBOL's support of features in the next ANSI COBOL standard, of the VAX Information Architecture, and of other Digital-defined extensions to COBOL makes possible a wider range of COBOL applications on the VAX.

*by B. Kernighan and D. Ritchie

VAX DIBOL - VAX DIBOL (Digital Interactive Business Oriented Language) is a high-level, procedural language designed specifically for interactive data processing in the business environment. It takes full advantage of the VMS system's facilities.

VAX DIBOL is based on the DIBOL Standards Organization's DIBOL-83 definition of the language. VAX DIBOL is highly compatible with DIBOL-83 implementation on other operating systems.

VAX DIBOL consists of a DIBOL compiler, a sharable runtime library, a program debugging aid called the DIBOL Debugging Technique (DDT), and a set of utility programs that facilitate data handling, data storing, and interprogram communication.

VAX DSM - VAX DSM (Digital Standard MUMPS) is a user data management and language processing system. The DSM language is a high-level, interpretive language well suited for the processing of variable-length string data. It conforms to the American National Standard MUMPS specification X11.1-1977.

VAX FORTRAN - VAX FORTRAN is a high performance, industry-leading implementation of the FORTRAN language. VAX FORTRAN is based on the American National Standard FORTRAN X3.9-1978 (commonly called FORTRAN-77). The VAX FORTRAN compiler supports this standard at the full-language level. Also, it provides full support for many industry-standard FORTRAN features based on FORTRAN-66, the previous ANSI standard. The qualifier /NOF77 will select the FORTRAN-66 behavior where the two standards conflict.

VAX LISP - For the past 20 years, the LISP programming language has been associated with artificial intelligence (AI) research. The LISP ("LIS"t "P"rocessing) programming language is based on a paper, published in 1958 by John McCarthy, dealing with non-numeric computation. It differs from the majority of higher-level programming languages in that LISP programs do not use numeric computation as a basis for program execution (although it does support facilities for numeric computation).

LISP is particularly useful for the manipulation of symbolic data. Symbols can be thought of as words; lists of symbols are then equivalent to sentences or statements. Because LISP's symbolic processing and knowledge representation capabilities can be used to represent human thought patterns and associations, the LISP programming language has become an essential tool for researchers as they attempt to make computers simulate human behavior and thought.

VAX Pascal - VAX Pascal is a multipass, optimizing compiler that is a powerful superset of the Pascal language as defined by Jensen and Wirth in Pascal User Manual and Report (1974). VAX Pascal accepts programs that are compatible with either the ANSI/IEEE 770X3.97 standard or the ISO standard (DIS 7185).

Pascal's block structured nature, flexible data types and English-like statements result in significant ease-of-use benefits. These benefits include ease of program generation and ease of reading, modifying, and maintaining programs.

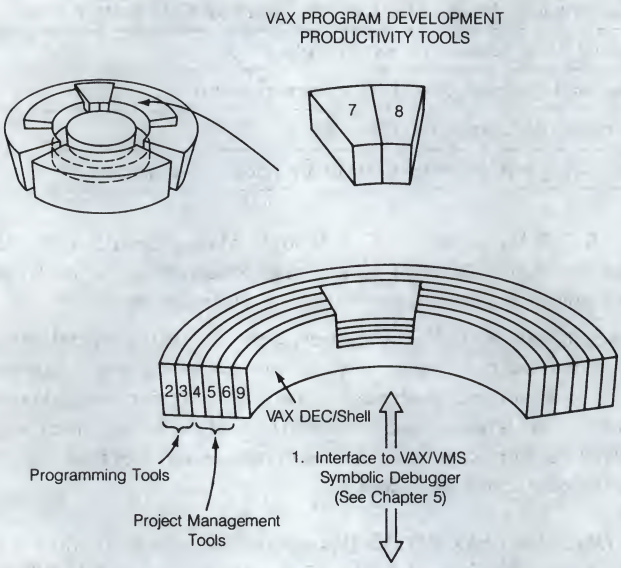
VAX PL/I - VAX PL/I is an extended implementation of the General Purpose Subset (X3.74-1981, "Subset G") of ANSI PL/I, X3.53 -1976. PL/I was designed to be useful in scientific, commercial, and system programming, especially on small and medium-sized computer systems. The goals of the design of Subset G were to include features that are easy to learn, easily portable from one computer system to another, to exclude seldom used features that increase runtime complexity.

VAX RPG II - RPG (Report Program Generator) is a powerful, business-oriented language specifically oriented to generating a wide variety of simple and complex business reports. RPG is a partially nonprocedural language, and is therefore not suited to all business applications. However, where it's appropriate, RPG can significantly increase your productivity and greatly improve turn-around time for generation and file maintenance application development cycles.

RPG II is an enhanced version of RPG, which was developed by International Business Machines Corporation in the early 1960s. RPG II incorporates a wide variety of additional features not present in the original RPG, and provides extra advantages in simplicity, ease-of-use, and power. RPG II has become a popular and widely used business application language. It is an important language for many small business users.

- VAX PRODUCTIVITY TOOLS

  VAX productivity tools help you write, compile, link, and maintain systems and applications designs by addressing multiple phases of the software development life cycle. For more on VAX Program Development Productivity Tools, turn to Chapter 2 of this handbook.



PROGRAMMING TOOLS

1. VAX/VMS Symbolic Debugger
2. Language-Sensitive Editor
3. Performance and Coverage Analyzer (PCA)

PROJECT MANAGEMENT TOOLS

4. DEC/Code Management System (CMS)
5. DEC/Module Management System (MMS)
6. DEC/Test Manager

GRAPHICS TOOLS (For device-independent graphics application programs)

7. GKS
8. DECOR

Figure 1-6 ▪ VAX Program Development Productivity Tools

VAX DEC/CMS - VAX DEC/Code Management System (CMS) is a program librarian for the development and evolution of application software in the VAX/VMS Software Development Environment. It comprises a set of commands that enables software developers to cooperatively manage the files of ongoing projects. A few of the many operations CMS performs are

- Storing ASCII files in a project library.
- Maintaining a history of all library modifications.
- Managing concurrent modifications.
- Identifying and "freezing" software for release or milestones.

VAX DEC/MMS - Digital's VAX Module Management System (MMS) is designed to manage "system builds" for developers during day-to-day development, implementation, and maintenance of a software system.

In the development of a large software system, many of the dependent modules, of which the system is made, are typically in various states of completion. VAX DEC/MMS determines which modules need to be recompiled, and ensures the software system is recompiled and linked with all the latest changes. VAX DEC/MMS can also interact with VAX DEC/CMS and extract files from CMS libraries when building a software system.

VAX DEC/Shell - VAX DEC/Shell is an optional software product that allow programmers, familiar with UNIX V7, to operate in the VAX/VMS Software Development Environment with a command language and utilities based on the Bourne Shell.

When using the DEC/Shell, you are not just limited to its commands and utilities. All the power and features of the Digital command language (DCL) are also available. For example, you can use DEC/Shell to write a command procedure with "Shell" and DCL commands. One important feature of the DEC/Shell that allows you to mix commands in pipelines.

The three major components of VAX DEC/Shell are the command-line interpreter, utilities, and the Shell script language.

VAX DEC/Test Manager - Testing is a necessary and costly part of software development. VAX DEC/Test Manager manages the testing process to help you produce higher-quality products with less maintenance.

VAX DEC/Test Manager is an automated regression testing system. That is, it automatically executes user-defined tests on existing software products and compares the test output against the product's benchmarks to determine if the software is performing as expected. It gives you the flexibility to organize and select tests for execution, run those tests, and in verifying and review the results.

VAX DEC/Test Manager makes software maintenance more manageable and helps you during the implementation of an application program. You can insert specific tests into a group and to then call those tests (or those of other programmers) independently.

VAX Language-Sensitive Editor - The VAX Language-Sensitive Editor is a multilanguage, multiwindow, screen-oriented editor designed specifically for program development. It is "language sensitive" in that it provides you with information on all the VAX languages it supports. It gives you full access to detailed templates of all of the major constructs in each of these languages.

The VAX Language-Sensitive Editor works with many VAX languages and the VAX Debugger and VAX Performance and Coverage Analyzer. Within a single editing session you can write code, edit, compile, and review and correct compilation errors. You can customize and extend the editor to meet your unique programming needs.

VAX Performance and Coverage Analyzer - The VAX Performance and Coverage Analyzer is a program development tool for measuring and tuning the performance of user-mode applications programs. It also measures test coverage, showing that routines, lines, or even individual code paths in a program are executed by a given suite of test input.

The VAX Performance and Coverage Analyzer consists of two parts: A Collector that collects performance or coverage data from a running program, and the Analyzer that later processes the data to produce various reports. The Collector can collect program counter sampling data, page fault data, system service counts, I/O usage data, exact execution counts at specified locations, and test coverage data. The VAX Performance and Coverage Analyzer is fully symbolic and can be used with all languages that have VAX Debugger Support.

VAX GKS - VAX GKS is a productivity tool for developing device-independent graphics applications. VAX GKS is ideally suited for writing applications to be used in automotive, CAD/CAM, chemical, educational, environmental, and scientific areas.

With VAX GKS, you can create one application program for many different graphics I/O devices. No longer do you have to recode applications every time a new device is incorporated into your computing environment.

VAX GKS is based on the ANSI and ISO graphics standards. VAX GKS, Level 0b, provides basic output and input capabilities.

VAX DECOR - VAX DECOR is a graphics subroutine package that provides an interface between an application program and graphics devices. The interface is device-independent and supports user-developed device handlers, as well as those supplied with VAX DECOR.

VAX DECOR is Digital's implementation of the SIGGRAPH/ACM Graphics Standards Planning Committee's CORE proposal. It provides a device-independent interface between an application program and supported graphic devices.

- RELATED VAX PROGRAM DEVELOPMENT PRODUCTS

These products include VAX information management capabilities, which are needed for the implementation of database applications, for example.

- VAX Common Data Dictionary (CDD)
- VAX Database Management System (DBMS)
- VAX Relational Database Management System (Rdb/VMS)
- VAX Datatrieve
- The VAX Terminal Data Management System (TDMS)
- The VAX Forms Management System (FMS)
- The VAX Application Control & Management System (ACMS)

Products used to develop applications for other target systems include

- VAXELN
- VAX-11 RSX
- MicroPower/Pascal-VMS

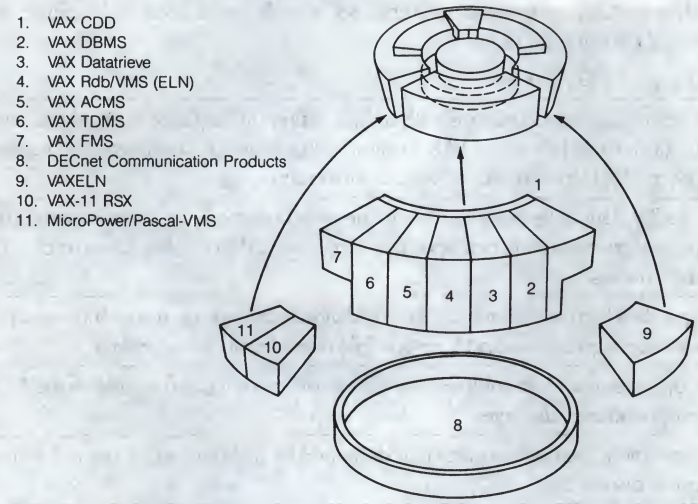


Figure 1-7 ▪ Related Program Development Products

VAX ACMS

VAX ACMS, the VAX Application Control and Management System, was designed to reduce the lifecycle costs involved in designing, developing, maintaining and controlling transaction processing and other complex VAX/VMS applications

Unlike traditional application development tools, ACMS allows for the replacement of large amounts of application code with high level definitions stored in the VAX Common Data Dictionary. With the use of such definitions, users now have available a fourth generation-like language facility that can significantly reduce the development and maintenance lifecycle costs of large, complex software projects.

For more on VAX ACMS, see the *VAX Information Management Handbook.*

VAX COMMON DATA DICTIONARY (CDD) - The CDD is a central location for the storage of data definitions used by many VMS software products. The CDD is required when using VAX DATATRIEVE, VAX Rdb/VMS, VAX TDMS, and the VAX ACMS product set. The CDD can be used in conjunction with most VAX programming languages, which access record definitions in the CDD at compile time.

Using the CDD you can

- Create sharable definitions with a data definition language (CDDL) that can be understood by many VAX programming language compilers and several other VAX Information Architecture Products.
- Modify data definitions in the dictionary without editing the programs and procedures using the definitions, as long as you don't delete fields or change their names.
- Specify which users have access to individual definitions, using thirteen separate access privileges and four possible user identification criteria.
- Copy definitions at compile time into a program written in one of the VAX programming languages.
- Document the use of a particular definition by making entries into the definitions' history list.
- Maintain an area of the dictionary for the storage of data definitions for privileged use.

For more on the CDD, see the *VAX Information Management Handbook.*

VAX DATABASE MANAGEMENT SYSTEM (DBMS) - VAX DBMS is Digital's CODASYL-compliant database management system. VAX DBMS provides sophisticated capabilities for creating, accessing, and maintaining large databases. The major advantages of VAX DBMS are its efficiency and its ability to control the sharing of the same data by a large number of users, all using the system at the same time.

VAX DBMS is also compatible with other VMS software products. You can access a VAX DBMS database from any application program written in a high-level language that conforms to the VMS calling standard. This is accomplished through an interface to VAX DBMS, called Database Query (DBQ).

With VAX DBMS, you can

- Create and maintain multiple databases.
- Separate data definitions from the applications programs that use them.
- Centralize all data definitions from the applications programs that use them.
- Define useful relationships between records.
- Separate data definitions from data storage.
- Tailor user views of data.
- Centralize the administration of data.
- Maintain data integrity and security.
- Allow concurrent access to databases.

For more on the DBMS, see the *VAX Information Management Handbook*.

VAX RELATIONAL DATABASE MANAGEMENT SYSTEM (VAX Rdb/VMS) - VAX Rdb/VMS provides facilities for creating, accessing, and maintaining relational databases. In a relational database, data is stored in the form of two-dimensional tables, instead of in the form of complex hierarchies or networks. VAX Rdb/VMS provides all the advantages of a full-feature database management system, including data security and optimized data access. At the same time, it provides easy -to-use features inherent in a relational-style database.

VAX Rdb/VMS is easy to use and understand. With it, you maintain and create a database without the services of a professional database administrator.

Two VAX Rdb products are currently available — VAX Rdb/VMS and VAX Rdb/ELN. VAX Rdb/VMS runs on the VMS operating system; VAX Rdb/ELN, on the VAXELN system.

With VAX Rdb/VMS you can

- Create and maintain relational databases.
- Store and retrieve data.
- Separate data definitions from the application programs that use them.
- Store all data definitions either in the database files or in the VAX Common Data Dictionary (CDD), for common use and maintenance.
- Establish dynamic relationships between records.
- Tailor user views of data.
- Centralize the administration of data.
- Maintain data integrity and security.
- Use a database concurrently with other users.
- Ensure against data inconsistency during concurrent updating.
- Recover from errors and/or inadvertent terminations.

For more on the Rdb/VMS, see the *VAX Information Management Handbook.*

VAX DATATRIEVE - VAX DATATRIEVE is an easy-to-use tool for managing and manipulating data either interactively at a terminal or from an applications program. If you know as few as ten simple, English-like commands and statements, you can interactively retrieve, store, modify, and sort data, and report on it in meaningful ways. With VAX DATATRIEVE, you can

- Create data definitions that can be used to store and retrieve data uniformly, either interactively or from application programs
- Store or modify data in RMS files, VAX DBMS databases, or VAX Rdb/VMS databases.
- Retrieve data from RMS files, VAX DBMS databases, or VAX Rdb/VMS databases and display the data on a terminal, write it to a data file, or print it.
- Produce formatted reports using specified selections of data display or data collection.
- Create pie charts, bar and line graphs, and plots based on specified selections of data.
- Use forms to format the terminal screen for data display or data collecting.
- Use a text editor to correct syntax errors and typing mistakes.
- Access data files located on remote systems.
- Call VAX DATATRIEVE functions (including data access) from a program written in a high-level VAX programming language such as COBOL, FORTRAN, or PL/I.

For more on VAX DATATRIEVE, see the *VAX Information Management Handbook.*

VAX TDMS - Digital's VAX Terminal Data Management System (TDMS) is a productivity tool designed to reduce the high life cycle costs of developing and maintaining forms-intensive terminal applications on VAX/VMS systems.

TDMS offers you a wide range of features that make it easy to develop applications to display and collect information. TDMS also eliminates many of the burdens associated with conventional forms-based application design and implementation.

TDMS provides an interface for defining the data exchange between screen(s) and program(s). It replaces the often cumbersome coding of program/screen interaction with definitions, which are stored independently in the VAX Common Data Dictionary.

For more on VAX TDMS, see the *VAX Information Management Handbook.*

VAX FMS - The VAX Forms Management System (FMS) is designed to aid in the development of application programs that use video forms. FMS manages these forms for application programs that use Digital's family of VT100 and VT200 compatible terminals. Forms defined using VAX FMS provides you with the following features of those terminals.

- Individual character attributes of reverse video, bold, blinking, and underline.
- Line attributes of double-width, double-height, and scrolling.
- Screenwide attributes such as 80- or 132-column lines and reverse video.
- Alternate character sets including the VT100 special graphics character set for line drawing.

For more on VAX FMS, see the *VAX Information Management Handbook.*

VAXELN - The VAXELN Toolkit offers you a unique alternative to the complex productivity problem of realtime application development. The VAXELN Toolkit lets you program in an extended version of Pascal or C, with all the high-level, easy programming features found in both the VAX languages.

The VAXELN toolkit is an optional product that supports the development of standalone, statically defined software systems (VAXELN systems) that run on the entire range of VAX processors, including the MicroVAX supermicrocomputer. VAXELN systems are developed under VMS, then run on the target system as a standalone system, without VMS present.

Typical applications include industrial automation, Ethernet server networks, and networks in which individual processors have dedicated functions and are not needed simultaneously for general computing, for which a general operating system, like VMS, is not necessary.

For more information on VAXELN, see Chapter 6 of this handbook.

VAX-11 RSX - VAX-11 RSX is an emulator for RSX-11 operating system family and executes on all VMS and MicroVMS systems. It runs in compatibility mode on processors that support a PDP-11 instruction set subset in hardware or microcode and it also runs on certain processors without this support by providing its own software emulation of the same PDP-11 instruction set subset. It provides special capabilities that enable you to develop programs for execution in any of the following environments:

- VAX/VMS compatibility mode
- MicroVAX II/MicroVMS (software-emulated compatibility mode)
- VAXstation II/MicroVMS (software-emulated compatibility mode)
- RSX-11M-PLUS
- RSX-11M
- RSX-11S
- Micro/RSX
- P/OS

VAX-11 RSX also allows for the migration of many existing RSX-11 applications to VAX/VMS and MicroVMS.

For more information on this product, see Chapter 6 of this handbook.

MicroPower/Pascal-VMS - MicroPower/Pascal-VMS is a VAX/VMS optional program development product. MicroPower/Pascal is a modular executive and software development package for PDP-11 (Q-bus) based microcomputer applications. It includes software components needed to create, build and debug/test concurrent realtime application software that runs standalone on a target runtime microcomputer system.

For more information on this product, see Chapter 6 of this handbook.


8
9
O
P

Chapter 2 ▪ VAX Program Development Productivity Tools

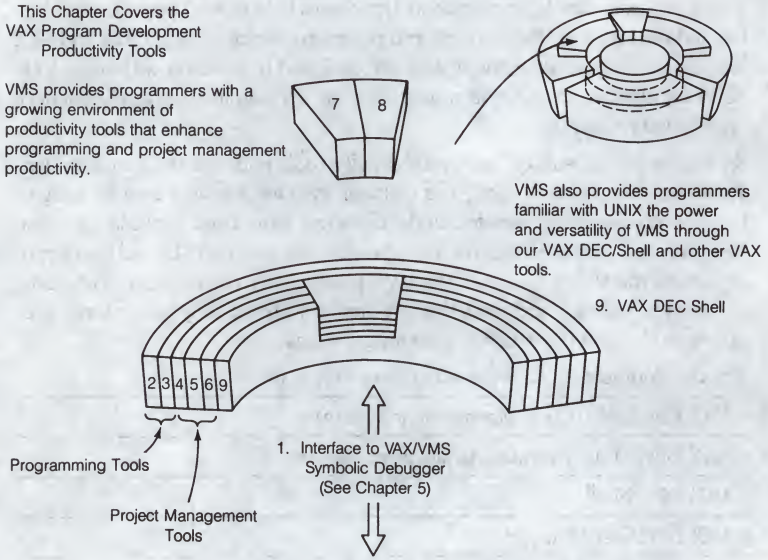
▪ Overview

VMS's program development productivity tools help extend your productivity beyond the range of VMS services and program development utilities and VAX languages. Because all of these tools are designed to function within the VMS Software Development Environment, they greatly simplify the development of application programs.

As Figure 2-1 illustrates, VAX productivity tools enhance the programmer's ability to write, compile, link, and maintain systems and applications designs. Besides serving as the basis for all development work done by VMS programmers, a subset of these tools, the VNX product set, give ULTRIX and UNIX programmers the ability to access the VMS productivity environment with many commands, utilities, and tools that are similar to those they use to write programs on the UNIX or ULTRIX operating systems.

The development tools discussed in this chapter are:

- VAX DEC/CMS (Code Management System)
- VAX DEC/MMS (Module Management System)
- VAX DEC/Shell
- VAX DEC/Test Manager
- The VAX Language-Sensitive Editor
- The VAX Performance and Coverage Analyzer
- VAX GKS



PROGRAMMING TOOLS

1. VAX/VMS Symbolic Debugger
2. Language-Sensitive Editor
3. Performance and Coverage Analyzer (PCA)

PROJECT MANAGEMENT TOOLS

4. DEC/Code Management System (CMS)
5. DEC/Module Management System (MMS)
6. DEC/Test Manager

GRAPHICS TOOLS (For device-independent graphics application programs)

7. GKS
8. DECOR (Not covered in this chapter)

Figure 2-1 ▪ Overview Of Chapter 2

▪ VAX DEC/CMS (Code Management System)

VAX DEC/CMS is a program library system that members of a software project use as an aid in program organization, development, and maintenance. CMS is a tool that allows project members to do the following with minimal change in work habits:

- Gain a more comprehensive view of project development
- Continue to maintain early versions of programs while working on current project development
- Communicate project information more easily
- Control multiple versions of the same software
- Have a complete history of project software

Designed to run under the VMS operating system, CMS provides a method of storing ASCII files. CMS maintains these files in project libraries. A library is a subdirectory. Once a library has been built, project members can retrieve copies of files from the library, work on them using any standard editor, and then replace them in the library. CMS keeps the original versions of the library files and integrates any subsequent changes. CMS also allows more than one project member to work on the same file at the same time without losing the modifications made by any project member.

As project development continues, CMS tracks changes to a project library by storing the changes made with each retrieval and replacement file in the library. As a result, CMS can reconstruct any previous version of a project file.

In addition to storing successive changes to library files, CMS monitors library access. CMS enhances communication among project members by maintaining both a record of who is currently working on a library file and a historical record of library access. By entering CMS commands, project members can easily retrieve information about library transactions and contents.

VAX DEC/CMS Features

You can perform the following operations with CMS commands:

- Put a file in a library
- Retrieve a file or files from a library for modification
- Retrieve a file or files from a library for read-only purposes
- Group any selection of files to access them together
- Reconstruct any milestone of a project by getting the versions of those files used in the construction of that milestone
- Determine whether any other user is working on a file before you retrieve it
- Place library files into "classes" at any phase of development to represent milestones
- Merge concurrent modifications to a file
- Obtain historical information about all transactions that update the library together with transactions that involve retrieving read-only copies of library files, and the transactions associated with a particular file
- Obtain information that describes the organization of library files
- Obtain listings of any file stored in the library that shows:
  - any stage of development of the file itself
  - the file history
  - when each line of changes were incorporated into the file and by whom
- Compare two ASCII files in your working directory or library

CMS also has a callable interface available. CMS provides an on-line help facility for information on library structure, commands, syntax, and other relevant topics.

Using VAX DEC/CMS

To create a CMS library, you need to create a subdirectory, and the CMS CREATE/LIBRARY command. Once your library has been created, use the INSERT command to put copies of your programs (files), documents, and/or tests into the library. These files are termed "elements: and can be managed by the use of various CMS commands. For an example of how CMS is used in the implementation of a software program, see Chapter 7 of this handbook.

When you want to modify an element, you use the CMS RESERVE command to retrieve a copy of the element file from the library. When you reserve an element from the library, CMS places a copy of the element in your default directory and marks the library copy as reserved by you. As long as you have the element reserved, CMS notifies any user who tries to reserve, retrieve, or replace that same element that you are working on it. (If you do not intend to modify an element, CMS can retrieve an element without reserving it.)

You can use any standard editor to work on files that you have reserved. When you have finished working on an element, you replace it in the library with the CMS REPLACE command.

Each time you create an element by loading a file into a CMS library, CMS creates the first generation of the element. Figure 2-3 is a symbolic representation of a newly created element, TEST.FOR.

Every time you reserve and then replace the element, CMS creates a new element generation and gives it a unique generation number. CMS can store up to 600 generations of an element. CMS is an efficient file storage system because it keeps only the differences between successive file versions. Lines that are duplicated from generation to generation are not stored. Figure 2-2 illustrates element TEST.FOR after one reservation/replacement cycle.

CMS allows access to any generation of an element. Because CMS keeps track of file modifications according to the succession of generations, you can reserve any generation of an element.

- CONCURRENT DEVELOPMENT

  CMS handles concurrent development of the same element by allowing you to establish alternate development paths, called lines of descent, for an element. CMS maintains the separate lines of descent and allows you to merge them when necessary during development. Generation 1 and 2 of element TEST.FOR (Figure 2-2) are on the main line of descent.

  Because a concurrent reservation might result in conflicting changes, CMS does not allow more than one user to replace a concurrently reserved element on the same line of descent. In this case, one user can replace the element and create a new generation on the same line of descent. Each other user who has this generation reserved must create a variant generation. Figure 2-2 shows element TEST.FOR after two users have concurrently reserved and replaced it.

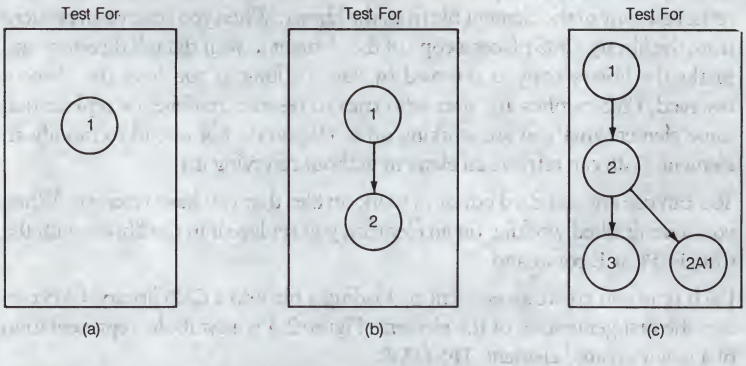


Figure 2-2 ▪ Element TEST.FOR with Main Line and Variant Generation

You can merge any two generations that are not on the same line of descent. When you use the CMS RESERVE command with the /MERGE qualifier, CMS merges the two named generations and from them creates a file in your default directory. CMS flags any conflicts between the two lines of descent. Once you have resolved the conflicts, if any, you can replace the element in the library. Figure 2-3 shows two lines of descent that have been merged. After testing, the merged copy was then replaced.

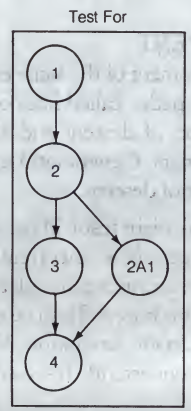


Figure 2-3 ▪ Generation 2A1 Merged into the Main Line of Descent

- CLASSES AND GROUPS

  CMS allows you to associate an element generation with others in sets called classes. For example, you can establish a class named RELEASE1 that includes generation 3 from one element, generation 5 from the second element, and so on. An element generation can belong to more than one class. Thus you can

establish classes that reflect milestones in project development. You can redefine classes for maintenance purposes, even after development has proceeded beyond the milestone. As an alternative, you can freeze a class so that it cannot be redefined.

CMS allows you to associate simple elements with others in sets called groups. For example, you can establish a group named DRIVERS that includes all of the device driver code for your project. A group named utilities might contain all the project utility routines. A group named SAMS could just be all of Sam's code.

- PROJECT INFORMATION

  CMS provides commands that give information on project history, status, or library contents. You can direct the output of these commands to a terminal, a file, or a lineprinter.

  CMS maintains a project history that is a record of all transactions that have updated the library and transactions that have retrieved elements for reading purposes only. Many transactions allow you to enter a remark to describe the reason for using the library. CMS records the remark along with the data, time, user, and command issued.

  The CMS library history provides a record of:

  - Transactions that created specific element generations
  - Transactions related to the evolution of a specific element
  - The entire transaction history for the library

  You can use CMS commands to create a report that integrates historical information into an element.

- **VAX DEC/MMS (Module Management System)**

  VAX DEC/MMS is a tool that automates and simplifies the building of software systems. MMS is useful for building both simple programs, which may have only one or two source files, and complex programs, which may consist of several source files, message files, and documentation. It can rebuild all the components in a system, or only those that have changed since the system was last built. MMS is also easy to use — with one command, you can build either a small or a large system.

MMS handles all the steps that are necessary to build a software system accurately and economically.

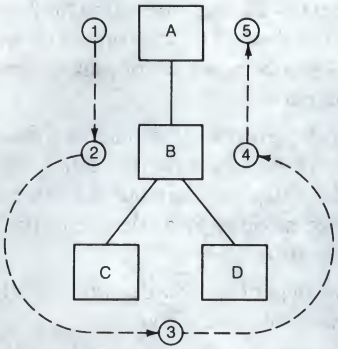
MMS runs on the VMS operating system and can be used

- With any VMS-supported compiler.
- To access modules in VMS object-module libraries.
- To access elements in DEC/CMS libraries.
- To access declarations of Forms stored in the VAX CDD (Common Data Dictionary).
- To access forms in VAX FMS (Forms Management System).
- To build documents.
- To build individual programs or entire systems.

VAX DEC/MMS Features

MMS controls the efficient building of a software system by determining which components in the system have changed and then creating new versions of only those files that depend on the changed components.

Figure 2-4 depicts a small software system and describes the basic steps MMS follows when it builds the system. In this system, Component A is the target — file that you want to update. Component B is a source for Component A, and Components C and D are sources for Component B. (For example, A might be an .EXE file, B might be an .OBJ file, and C and D might be source or definition files). The commands that update B (by using C and D) and A (by using the updated B) are called actions.



① MMS checks the revision time of the target (Component A).

② MMS checks the revision time of the first source (Component B).

③ MMS checks the revision times of Components C and D against that of B.

④ If the times of Components C and/or D are more recent than that of Component B, MMS updates B according to *action lines* that you specify in the description file (action lines tell MMS what commands to execute to update the components of a software system). If B is more recent than C and D, MMS does not do anything to B because it is already up-to-date.

⑤ Once Component B is updated, it is more recent than the target. Therefore, MMS updates Component A.

Figure 2-4 ▪ How MMS Builds a Software System

If the target has been modified since the sources were last changed, MMS does not take any action to update the target. Instead, it issues a message to inform you that the target is already up-to-date.

Using VAX DEC/MMS

When you use MMS to build your software system, you usually perform two steps:

1. Create a description file
2. Invoke MMS

The description file contains rules that describe how the components of your system are related, and the commands that MMS is to use in building them. Once you have created your description file, you can use it every time you invoke MMS to build your system. In most cases, you will need a description file; however, if your system is very simple, MMS can build it even if you have no description file.

The description file is an ASCII text file that you can create and modify with any text editor. It contains all the information MMS needs to update your software system. You can think of the description file as a collection of rules and instructions that describe how parts of your system are built, and which parts depend on other parts.

The order in which MMS processes the rules in a description file does not depend on the position of the rules in the description file. The order is implied because the output of one step may the input of another step. For example if a link step uses three object modules, then MMS cannot process the link step until all three object modules are up-to-date.

You can leave out some steps of an MMS description file because MMS built-in rules define certain commonly used actions.

When you invoke MMS, it looks in your current directory for a description file called DESCRIP.MMS, unless you have specified the /NODESCRIPTION qualifier to indicate that no description file is to be used. Once it locates the description file, MMS processes it.

MMS allows the "top-down" breakdown of a task. Thus, you can use mnemonic names at the beginning of a description file to specify the order in which tasks must be accomplished, and then specify the actions to accomplish those tasks further on in the description file. With MMS, the structural information is all in one place, the description file, giving a clear representation of the system.

- DEPENDENCY RULES

  A description file contains dependency rules. Dependency rules tell MMS the relationships among the files in the software system and specify the actions MMS is to perform to update those files.

  The general dependency rule format is shown in Figure 2-5.

```
TARGET(S) : SOURCE(S)
        ACTION LINE(S)
```

Figure 2-5 ▪ Dependency Rule Format

The target in a dependency rule is the file that is to be updated by MMS. The source is the file that MMS uses to update the target. (A source in one dependency rule can also be a target in another rule, if it needs to be updated itself before building its target.) The action lines contain commands that tell MMS how to update the target using the source.

Figure 2-6 shows how the relationship between an object file and a BASIC source file is expressed as a dependency rule.

```
TEST.OBJ : TEST.BAS
        BASIC TEST
```

Figure 2-6 ▪ Sample Dependency Rule

The dependency rule in Figure 2-6 tells MMS that it should execute the BASIC command to create TEST.OBJ from TEST.BAS.

You usually have more than one dependency rule in a description file. The number of rules depends on the size of your system. When MMS processes a dependency rule, it uses the last revision dates of the target and source files to determine whether the target should be updated. (The revision date indicates the date and time the file was most recently changed.) If you have not changed any of the source files since the last time the target was built, MMS does not execute the dependency rule action line(s). If, however, you have changed any of the source files, MMS executes the action line(s) to make the target up to date. MMS includes several default dependency rules, called built-in rules, that allow MMS to automatically build targets that depend upon sources written in VAX/VMS-supported languages. In addition, you can write your own user-defined default rules.

Compatibility in the VMS Environment

You can use MMS to access files that are contained in VMS object-module libraries. You can also use MMS to create library files, using certain built-in rules that apply only to VMS libraries.

If VAX DEC/CMS (Code Management System) is installed on your system, you can use MMS to access elements in CMS libraries. MMS provides built-in rules that apply only to CMS library access. From the MMS command line, you can control whether MMS automatically looks for elements in a CMS library when it processes a description file. You will probably want to keep your description file in a CMS library so you can keep track of the changes in dependencies of your system. MMS can access the correct generation of the description file in a CMS library.

If the VAX CDD (Common Data Dictionary) is installed on your system, you can use MMS to access records stored in the CDD. MMS provides built-in rules that apply only to CDD record access. If VAX FMS (Forms Management System) is installed in your system, you can use MMS to access forms stored in FMS libraries. To specify an FMS form in a dependency rule, use the file type .FLB after the library name. The default file type for FMS forms is .FRM.

For an example of using VAX DEC/MMS in the implementation of a software program, see Chapter 7 of this handbook.

▪ VAX DEC/Shell

If familiar with a UNIX program development environment, VAX DEC/Shell gives you the capabilities of the Bourne shell and many utilities from the UNIX V7 system. VAX DEC/Shell provides an alternative command line interface to the VMS operating system.

VAX DEC/Shell:

- Gives you more than 50 of the most commonly used UNIX utilities
- Is not a UNIX emulator on VMS. It is integrated with VMS. DEC/Shell and VMS commands may be freely intermixed, even on the same command line.

There are three major components of the DEC/Shell: the command-line interface, the Shell script language, and common utilities. When combined, these components provide a program development environment familiar to users experienced with UNIX V7 system.

VAX DEC/Shell Features

DEC/Shell includes:

- The UNIX V7 Bourne Shell
- A large number of UNIX utilities
- Pipes
- Input/output redirection to and from files
- A UNIX file syntax emulator
- A Shell runtime library

DEC/Shell also provides access to DCL commands and VMS programs. This capability allows users familiar with the UNIX V7 system to take advantage of the VMS operating system while working in a familiar programming environment.

Using the VAX DEC/Shell

You can invoke the Shell from the DCL level or you can make the Shell your default command interpreter when you log in.

To invoke the Shell from DCL, you use the SPAWN command with the /CLI qualifier:

```
$ SPAWN/CLI_=SHELL
```

This command creates a subprocess with the DEC/Shell instead of DCL as the command interpreter. In this subprocess, you can perform many of the tasks you would do on a UNIX V7 system.

To make the Shell your command interpreter when you log in, you can type /CLI_=SHELL/NOCOMMAND after your name at the username prompt when logging into the system (Username: SMITH/CLI_=SHELL/NOCOMMAND, for example), or ask the system manager to make DEC/Shell the default command interpreter for your account.

The VAX DEC/Shell as a Programming Language

While the DEC/Shell can be used primarily as a command interpreter, it is also a powerful programming language. Many of the control structures used in the Shell are similar to those used in the C Language (See Chapter 3 of this handbook). Given the Shell script language, control-flow constructs, and utilities, you may find that the DEC/Shell is an adequate language for many of your programming needs.

The VAX DEC/Shell Runtime Library

The DEC/Shell includes a runtime library that has routines for UNIX and VMS file name conversion, along with other general purpose routines. These routines are intended to minimize difficulties in transporting existing UNIX applications to VMS.

The Shell runtime library is provided as a convenience for those interested in transporting UNIX C programs to VMS. It does not provide UNIX call-level emulation, which is done by the VAX C language.

This section lists in alphabetic order the commands and utilities provided by the DEC/Shell and briefly describes the function of each.

Table 2-1 ▪ Utilities and Commands

Command	Function
awk	Performs actions when lines of files match specified patterns
basename	Removes file name affixes
cal	Displays a calendar
cat	Concatenates and displays files
cd	Changes your working directory
chmod	Changes protection modes of files
chown	Changes owners of files
cmp	Compares two files
cp	Copies files
date	Displays or sets the system date and time

(continued on next page)

Table 2-1 ▪ Utilities and Commands (Cont.)

Command	Function
dc	Performs arithmetic calculations
dcl	Invokes the DCL command parser; also provides a one-line escape to DCL
diff	Compares two files
diff3	Compares three versions of a file
echo	Echoes arguments on the standard output
ed	Invokes the UNIX line editor
eval	Expands all command-line arguments and executes the result as a command
exec	Executes a command and exits
export	Passes environment variable to programs and sub-processes
expr	Evaluates arguments as an expression
false	Returns a failure status value
find	Searches directory hierarchies for files
grep	Searches files for a pattern
join	Joins files according to specified relations
kill	Terminates processes
lex	Generates lexical analysis programs
login	Connects your terminal to another node on the network
logout	Terminates an interactive terminal session
ls	Lists the contents of a directory
m4	Processes macros
mcr	Invokes the monitor console routine (MCR) command parser
mkdir	Creates a directory
mv	Moves files and directories
od	Dumps files in octal format or other formats
pr	Formats and displays files
ps	Shows process and system status

(continued on next page)

Table 2-1 ▪ Utilities and Commands (Cont.)

Command	Function
pwd	Displays the name of your current directory
read	Assigns values from the next line read to the specified variables
readonly	Prevents variables from being reassigned
rm	Deletes files from directories
rmdir	Deletes directories
sed	Edits input streams
set	Modifies and/or displays Shell environment characteristics
sh	Invokes another Shell
sleep	Suspends execution for a specified interval
sort	Sorts and/or merges files
tail	Displays the last part of a file
tar	Saves and restores files on magnetic tape
tee	Writes standard input to standard output, making copies in files
test	Evaluates expressions
times	Reports the CPU time used by processes
touch	Updates the last modification dates of files
tr	Translates characters
trap	Specifies commands to execute upon receipt of signals
true	Returns success status values
tty	Displays the name of your terminal
umask	Sets and/or displays default file protection
uniq	Displays nonrepeated lines in a file
units	Converts quantities expressed in standard scales to their equivalents in other scales
wait	Waits for completion of processes
wc	Counts words, lines, and/or characters in files
who	Displays a list of logged-in users on the system
yacc	Generates LR(1) parsing tables

▪ VAX DEC/Test Manager

The VAX DEC/Test Manager is a tool that organizes software tests and automates the way you run tests and evaluate their results. It provides an efficient way to organize, run, and store the results of existing tests.

DEC/Test Manager is based on the concept of regression testing. Regression testing is a method of ensuring that a program being developed runs correctly and that new features added to a program do not affect the correct execution of previously tested features.

In regression testing, you run established software tests and compare the actual test results with the results you expected to get. If these actual results do not agree with what is expected, the software being tested may contain errors. If errors do exist, the software being tested is said to have "regressed."

Features of VAX DEC/Test Manager

You can use DEC/Test Manager to create a library area for test result storage. Test Manager allows you to:

- Create descriptions of software tests
- Group these tests descriptions into meaningful combinations for later runs
- Modify or display the test descriptions or groups
- Execute specific tests, groups of tests, or combinations of groups of tests
- Compare the results of each executed tests with its benchmark test results to determine differences
- View test results interactively
- Update benchmarks as needed

Using VAX DEC/Test Manager

The following two lists detail the steps you might follow in a typical regression testing procedure. The first describes the regression testing procedure without using DEC/Test Manager; the second describes the same procedure with DEC/Test Manager.

- STEPS IN REGRESSION TESTING

1. Create tests by writing command files to test your software.
2. Organize your tests.
   - Create a mechanism to allow ready access to tests as needed.
3. Run the test:
   - If you wish to run a single test, submit its command file to the batch queue.
   - If you wish to run a number of tests, create a command file that invokes each test, and submit it to the batch queue.
   - Examine the test results:
     — Compare the results to those you would expect. Note any differences between the expected and actual results.
     — For incorrect test outputs, revise the program code to correct the bugs. Repeat steps 3 and 4 until the test output is correct. Then save the validated results.
4. Repeat Step 3 whenever you modify the program, or add new code. Then, compare the current test outputs with the validated test outputs.
   - If the current and validated test outputs match, the program being tested is working correctly.
   - If you find unexpected changes in test results, the new program probably contains errors, or has "regressed."
   - Correct the program and rerun the tests whose outputs have not matched. Repeat this cycle until all results are valid. For future test runs, use these validated test outputs as references against which to compare the current test outputs.

DEC/Test Manager automates steps 2 through 4 described previously. When you use DEC/Test Manager, you must still write your own tests and test data, and create a command file that runs the test. However, DEC/Test Manager automates the rest of the testing procedure.

- STEPS IN REGRESSION TESTING WITH DEC/TEST MANAGER

1. Create tests by writing command files to test your software.
2. Organize a DEC/Test Manager system by
   - Creating a DEC/Test Manager library
   - Identifying each test and its related files to DEC/Test Manager
   - Placing tests into groups if you wish to categorize them
3. Run the test.
   - Use DEC/Test Manager to select the test or set of tests you want to run.
4. Compare current test results with the benchmark results for a test.
   - DEC/Test Manager automatically compares test results with the expected results (called benchmark files) for a test. It records any differences in a DEC/Test Manager-created differences file.
5. Examine test results.
   - DEC/Test Manager provides an interactive subsystem that lets you immediately access test results. It also gives you the ability to group all tests that gave incorrect results for easy retesting.
   - If the current and validated test outputs match, the program being tested is working correctly.
   - If you find unexpected changes in test results, the new program probably contains errors, or has "regressed."
6. Repeat steps 3 through 5 whenever you modify the program or add new code. Using DEC/Test Manager, the selection and running of tests and the evaluation of results will progress more quickly.

- ORGANIZING TESTS

DEC/Test Manager provides a number of structures for organizing a test system.

The first of these is the test description. A test description identifies a test to DEC/Test Manager. It consist of fields whose contents point to files needed to run the test.

The core of each test description is the template file. A template file is a DCL command procedure that runs a specified test or that is the test itself. Each test must have a template file.

You can optionally specify a test prologue and a test epilogue. These are command procedures that extend the test environment. A prologue is a setup file run immediately before the template. An epilogue can function as a cleanup file or as a filter for test results and is run immediately after the template.

A variable in DEC/Test Manager is either a DCL symbol or a logical name. You can use variables in templates, prologues, and epilogues. For example, by using a variable in place of a particular test name in a template file, you can use that same template file to run many tests. You must define variables to DEC/Test Manager in a separate step, and also include them in each test description that uses them.

You can categorize test descriptions by placing them in groups. For example, if you have several tests that share a similar characteristic or function, such as testing a parser, you can create a group called PARSER and place those tests in this group. A test can belong to more than one group.

- RUNNING TESTS

  DEC/Test Manager runs tests in batch mode. To run tests with DEC/Test Manager, you select the tests you want to run by group or by test description name and create a collection name for them to identify the tests as a set.

  You can supply a prologue or epilogue file for a collection at the time you create the collection. The prologue file is a command file that runs before the collection tests are run. The epilogue file is a command file that runs after the collection tests are run. You can also supply a default prologue or epilogue file that will be run with each collection.

  Once you create a collection, you submit it through DEC/Test Manager to a batch queue. DEC/Test Manager places the results of each test into an output file called the result file. The result file contains the output of the test execution at the time the collection was run.

  With DEC/Test Manager, it is possible to place a test in more than one collection, thus allowing concurrent use of tests.

  DEC/Test Manager automatically compares the result file for a test against its benchmark file. A benchmark file contains expected test output — that is, test output you have determined to be correct for purposes of the test. DEC/Test Manager stores the status of the comparison (successful or unsuccessful) along with any differences in the DEC/Test Manager library. If there are any differences between the result file and benchmark file, DEC/Test Manager creates a file called a differences file that stores the differences.

- EVALUATING TEST RESULTS

  You can evaluate test output files interactively with the DEC/Test Manager Review subsystem.

  DEC/Test Manager generates a result description for each test in a collection. A result description identifies the output files for each test and indicates the status of each test. It has the same name as its corresponding test description.

  Review allows you to

  - Display the status of each test.
  - Display or print test results of all related files.
  - Create a new benchmark or replace the old benchmark file by updating it with a result file.
  - Place tests in groups while viewing their results. This provides a convenient method of later rerunning tests.

▪ The VAX Language-Sensitive Editor

The VAX Language-Sensitive Editor is a powerful multilanguage, multiwindow, screen-oriented editor specifically designed for program development and maintenance. The Editor is "language-sensitive" in that it provides a description of the syntax for each VAX language that it supports. The Editor helps both novice and experienced programmers build syntactically correct programs faster and with fewer errors, through VAX language-specific construct completion, and error detection and correction facilities.

The VAX Language-Sensitive Editor is highly integrated into the VMS development environment. It is invoked via the English-like Digital Command Language (DCL) or the DEC/Shell command language, and works in concert with supported VAX languages and the VAX Symbolic Debugger to provide a highly interactive environment that facilitates the EDIT-COMPILE-DEBUG portion of the program development cycle. This enables users to create and edit code, and to compile and review, and correct compile-time errors in one streamlined editing session. The Editor can be invoked directly from the VAX Symbolic Debugger to correct source code errors found during a debugging session.

In addition, the VAX Language-Sensitive Editor offers you features to customize and extend your editing environment to meet any unique programming needs.

"Language-Sensitive" Features

The VAX Language-Sensitive Editor

- Tailors the editing sessions for each of the eight VMS languages it supports — VAX Ada®, VAX BASIC, VAX BLISS-32, VAX C, VAX COBOL, VAX FORTRAN, VAX PASCAL, and VAX PL/I.
- Uses formatted language-specific source code templates for quick, efficient source code entry.
- Allows compilation and review and correction of compile-time errors within a single editing session.
- Provides for interactive editing capabilities during a debugging session.
- Allows users to tailor the defined language environments or to define their own environment.

The VAX Language-Sensitive Editor interfaces with supported VAX languages to provide a powerful tool for program development. For each supported VAX language, the Language-Sensitive Editor provides a set of source code templates. These templates are formatted language constructs that provide keywords, punctuation, and placeholders. Templates are inserted into the editing buffer by successive expansions of tokens and placeholders. Placeholders represent positions in the source code where you must either provide additional program text or choose from indicated syntactic options. Tokens are keywords or function names that you can type into the editing buffer and expand into templates for corresponding language constructs.

While in an editing session, you can complete the editing of his code, compile it, and review the compile-time errors. You may specify DCL qualifiers such as /DEBUG and /LIBRARY when invoking the compiler from the VAX Language-Sensitive Editor. The compilation may be performed interactively or in a batch job.

The REVIEW command allows review of compilation errors upon compile completion. The VAX Language-Sensitive Editor displays the compilation errors in one window, with the corresponding source code displayed in a second window. For easy error correction, there is a GOTO SOURCE command to position the cursor at the point in the source code where the compiler detected the error.

The VAX Language-Sensitive Editor can be invoked from the VAX Symbolic Debugger providing the ability to make source code corrections as they are found during a debugging session. Features include:

- User notification if the file invoked by the editor is a different version than that displayed in the VAX Symbolic Debugger.
- Ability to specify the file and line number from which to start the editing session with the default being the current source displayed in the VAX Symbolic Debugger.
- User choice of terminating debugging session with the editing session or returning to the debugging session.

The VAX Language-Sensitive Editor enables users not only to customize previously defined language environments but also to define their own environments. Once they have defined their own environment, the Editor lets them save it in a file, restore it for later editing sessions, and update it with new definitions.

Using the VAX Language-Sensitive Editor

The Editor is invoked by the LSEDIT command. The language is determined from the file type of the file specification given in the command.

For the new user,

- The keypad layout is the same as for EDT.
- PF2 gives help on keys
- CTRL/Z allows you to enter line mode.
- CONTINUE or CTRL/Z allows return to keypad editing.
- There is a line mode HELP command.

When using templates, there are bracketed constructs that will be inserted into the source file. These are called placeholders. There are four commands of primary interest in inserting language constructs:

- CTRL/E — EXPAND
- CTRL/K — ERASE PLACEHOLDER /FORWARD
- CTRL/N — GOTO PLACEHOLDER /FORWARD
- CTRL/P — GOTO PLACEHOLDER /REVERSE

Depending on the type of placeholder at the cursor position, EXPAND will replace the placeholder with a sequence of keywords and/or placeholders, or it will present a menu of choices, or it will tell you what you have to enter as program text at that placeholder. A menu item is selected by using the arrow keys and then EXPAND. Text may be typed when the cursor is within placeholder brackets; the placeholder is automatically erased.

In addition to expanding placeholders, which are generally inserted for you, you may type in a template name and EXPAND it. Templates are provided for most keywords that introduce syntactic elements such as declarations and statements. For example, if you type in IF and then CTRL/E (EXPAND), an IF statement will be inserted at that point. It will contain placeholders for things that you must fill in. Templates are also provided for built-in functions and for the statement placeholder.

To compile the contents of the current buffer and review the errors, use the command COMPILE/REVIEW. This will display the errors in one window and your code in the other.

To select an error, position the cursor on that error and press CTRL/G (GOTO SOURCE). The cursor may be jumped forward from error to error using CTRL/F (NEXT ERROR) and backward using CTRL/B (PREVIOUS ERROR).

- ONLINE HELP WITH THE VAX LANGUAGE-SENSITIVE EDITOR
  When you want help, you want it immediately. And you don't want to sift through pages to find the information you need. In addition to the help you get by using the language-specific templates, the VAX Language-Sensitive Editor provides you with extensive online help facilities for each supported language.

  You can quickly access help on all placeholders, or call on language-specific online help topics directly from the Editor. If you are learning a new language, or are new at program development, you will gain efficiency as the Editor guides you with information on language structures and text insertion. If you are experienced in a language, this feature lets you access help on less familiar language constructs.

  Online help makes it easier for you to code it right, the first time.

- WINDOWS PROVIDE ADDED FLEXIBILITY
  The screen format for the VAX Language-Sensitive Editor is responsive to many different programming approaches. It consists of a prompt area, a message area, and one or two windows. Editor commands are used to manipulate the screen and its format.

You can create buffers to hold multiple files and can display any two buffers simultaneously. This feature lest you save development time by cutting and pasting sections of code or text from one buffer to another. Or you can designate READ ONLY buffers for version control protection. Since the Editor determines the language you are using by the filename extension, you can move between different languages in different buffers with the Editor providing the interfaces to the appropriate compiler.

To help you with particularly large programs where you might need to reference another section of the code, the Editor provides a split-screen capability. This feature enables you to view and work on two different sections of the same program simultaneously.

- TAILORING YOUR ENVIRONMENT
  The Editor's design allows you to use all of its powerful capabilities and still be able to incorporate any unique editing preferences you may have. You can create versions of the environment that you will work best in.

  For easy editing, you can bind any command or string of commands to a specific key. If you're presently a VMS user, you will quickly recognize the default keypad since it is similar to that used with EDT. Another important Editor feature lets you choose overstrike or insert editing mode.

  To match personal or specified coding conventions, you can modify the language-specific templates provided with the Editor. New templates may be added to those provided for the VAX languages, or you can designate a new language name to represent a set of templates you have created. For example, you can create a language called SPEC which contains templates, tokens and placeholders for structured, formatted specification documents.

  Once you have set up an editing environment to your specifications, you can save it and use it for future editing sessions — for the same project or for new applications.

- VAX TEXT PROCESSING UTILITY (VAXTPU)
  For more unique editing requirements, the VAX Language-Sensitive Editor provides commands to access the VAX Text Processing Utility (VAXTPU), a utility that is part of VMS. VAXTPU has an easy to use high-level procedural language which allows users to write functions not provided by the VAX Language-Sensitive Editor to further customize the editing environment. The VAXTPU language provides for looping and conditionals to allow users to perform more powerful editing tasks.

  For more on VAXTPU, see Chapter 5 of this handbook.

▪ VAX Performance and Coverage Analyzer (VAX PCA)

The VAX Performance and Coverage Analyzer is a VMS software development tool that helps you analyze the runtime behavior of your application programs.

The VAX Performance and Coverage Analyzer serves two functions:

- It pinpoints execution bottlenecks and other performance problems. Using this information, you can modify your programs to run faster.
- It provides you with test coverage analysis by measuring what parts of a user program are or are not executed by a given set of test data. Using this information, you can create tests that thoroughly exercise your programs.

By using the VAX Performance and Coverage Analyzer, you can focus your efforts on making improvements that will bring the biggest gains in a program's efficiency.

Features of the VAX Performance and Coverage Analyzer

The VAX Performance and Coverage Analyzer consists of two parts — the Collector and the Analyzer. The Collector gathers performance or test coverage data on a running user program and writes that data to a performance data file. The Analyzer — a separate, interactive program — then reads the performance data file and processes the data to produce performance histograms and tables.

VAX PCA can collect and analyze the following kinds of data:

- Program counter sampling data

  The program counter (PC) is sampled at a regularly specified interval (by default, every 10 milliseconds). This information provides a good overview of where your program is consuming the most time.

- Page fault data

  This information lets you determine what sections of the program are causing the most page faults.

- System services data

  This information tells you which system services the program calls, how often it calls them, and which sections of the program do the calling.

- Input/Output data

  This information about all Record Management Services (RMS) calls in your program can help you to understand your program's input/output behavior. This is essential when trying to speed up an I/O-bound application program.

- Exact execution counts

  This information about the exact number of times specified program locations are executed helps you to understand your program's dynamic behavior.

- Test coverage data

  This information shows you which code paths are or are not executed when you test your program. Using this data, you can make your test more complete.

Both the Collector and the Analyzer are fully symbolic, getting the required symbol information from the Debug Symbol Table (DST) generated by the compilers. Consequently, the VAX Performance and Coverage Analyzer works with any of the VMS languages that produce DST information. These include VAX Ada®, VAX BASIC, VAX Bliss, VAX C, VAX FORTRAN, VAX MACRO, VAX Pascal, VAX PL/I, and VAX RPG II.

Using the Collector

When using the Collector to collect performance or coverage data from a program, you must

1. Compile your program with the /DEBUG qualifier.
2. Link your program using the LINK/DEBUG_=SYS$LIBRARY:PCA$OBJ.OBJ command.
3. Run the program.

These steps ensure the necessary symbol table information is included in your image and that your program is linked with the Collector. When the program starts running, the Collector takes control and prompts you for commands.

Once you have invoked the Collector and gotten the Collector prompt (PCAC>), you are ready to enter Collector commands. First, you usually enter a SET DATAFILE command to specify the name of the performance data file, which will contain the collected performance data and all symbol information needed by the Analyzer. Then, you enter commands that specify what performance or coverage data you want to gather. Commands for gathering data include SET PC_SAMPLING, SET PAGE_FAULTS, SET SERVICES, SET IO_SERVICES, SET COUNTERS, and SET COVERAGE. Last, you enter the GO command to start the data collection.

When the GO command is entered the Collector starts your program, gathers the specified performance or test coverage data as the program runs to completion, and writes this data to the performance data file.

The following example shows the Collector commands to gather program counter sampling data for PRIMES:

```
PCAC> SET DATAFILE PRIMES.PCA
PCAC> SET PC_SAMPLING
PCAC> GO
```

When the program finishes running, the Collector closes the data file. You are then ready to run the Analyzer on the information in that file.

Additional collection runs can be made on your program by simply repeating the RUN command to initiate another run. It is also possible to collect data from many executions of the same program into a single data file.

Using the Analyzer

The Analyzer processes the data in a performance data file to produce histograms, tables, and other reports. The Analyzer lets you interactively examine the gathered data in various ways until you can pinpoint performance bottlenecks or identify code not covered by testing.

To run the Analyzer, you must use the PCA command at DCL level. This command accepts the name of a performance data file created by the Collector as a parameter. When you enter this command, the Analyzer product heading and prompt will appear on your terminal.

- ANALYZER PLOT AND TABULATE FEATURES

The two commands most frequently used in the Analyzer are PLOT and TABULATE. These commands let you present the information in the performance data file either as histograms (the PLOT command) or as tables of raw data counts and percentages (the TABULATE command). Both commands organize data in the same ways; the only difference between the two commands is in how they format the output.

The summary page at the end of every histogram or table gives various statistics and lists all qualifiers, node specifications, and filters used to generate the histogram or table.

- **VAX GKS**

The Graphical Kernel System (GKS) Standard specifies a set of graphics primitives or functions. The functions provide applications with a graphic system to produce two-dimensional pictures on vector or raster graphics output devices. The standard defines twelve upward compatible levels of functionality to meet the varying requirements of different graphic systems.

As an international standard, GKS provides a common definition for a graphics interface that implements functions common to all applications. Thus, GKS permits the applications programmer to concentrate on the job at hand, rather than on system level programming details. This increases programmer productivity and decreases the development time for a graphics application. GKS also makes an application portable among machines that implement the standard.

Features of VAX GKS

VAX GKS supports several types of output functions that draw the following components of a graphic image:

- Lines and Polylines
- Markers and polymarkers
- Text strings
- Polygons
- Rectangular arrays of pixels

Input functions provide the following general facilities:

- Initialization of logical input devices
- Acquisition of input from logical input devices
- Control of echoing and other characteristics of logical input devices

VAX GKS provides a method to file graphical information. A file of graphical information is useful for external storage, exchange, and reproduction of a graphic image.

Using VAX GKS

VAX GKS provides an interface between an application program and a collection of physical input and output devices. Each type of physical device is unique in capabilities and characteristics. Since VAX GKS is a device independent graphics system, the interface masks the low-level features of hardware device and communicates with all devices through a common interface that allows device-specific graphic handlers to perform the hardware instructions.

To provide a convenient and device independent coordinate system, you describe and construct graphic images using World Coordinate (WC) systems. World Coordinate systems permit the application programmer to define a range of values for the x-axis and y-axis that apply to a particular application rather than a specific device. For example, an application can use World Coordinates that range from -2 to 10 on the x-axis and from -6 to 6 on the y-axis.

In a WC system, the x-axis increases positively toward the right and negatively toward the left while the y-axis increases positively upwards and negatively downwards. The axes are always perpendicular and intersect at the (0,0) point, known as the origin. The position of any given point is simply the distance from the origin along the x-axis and the y-axis.

To compose a graphic image, you define the portion of the World Coordinate system that contains the points you wish to display. This portion of the World Coordinate system is known as the world window. After defining the world window, you specify the placement of the window onto a display surface.

Since the coordinates of a display surface are device dependent, VAX GKS defines an imaginary display surface on which to project the image. The imaginary display surface represents a single, square device independent display surface for all devices. This display surface is known as the Normalized Device Coordinate (NDC) space.

You can project the world window onto any portion of the NDC space. The portion of the NDC space that contains the world window is known as the world viewport.

A set of normalization transformations permits you to specify the world window and the world viewport limits.

The world viewport also defines a clipping rectangle. You can enable or disable clipping to the world viewport boundary (the clipping rectangle). With clipping enabled, points of an output function that fall outside the limits of the world viewport are not displayed. With the clipping disabled, VAX GKS draws the portion of the output whose locations extend beyond the limits of the world viewport.

- VAX GKS OUTPUT

  VAX GKS builds the display of a graphic image from two basic elements: output functions and output function attributes. The output functions are abstractions of basic operations a device can perform (for example, drawing lines and printing character strings). The output function attributes tailor the appearance of the object drawn by an output function.

Each output function is defined in a World Coordinate system. The output functions permit you to perform the following graphics operations:

- Draw lines
- Place markers
- Display text
- Fill a defined area
- Display an array of cells with individual colors
- Draw special, device-specific geometric shapes

For each output function, a set of output attributes describes the representation of the respective graphic image. For example, when you call the VAX GKS output function to draw a line (Draw Polyline), polyline attributes specify the form, thickness, and color of the line to draw. Initial values, supplied as default settings, display a general representation of the object drawn by each output function. You can change attribute values to tailor the display to application requirements.

VAX GKS provides three methods to specify output function attributes. The methods are:

- Individual
- Bundled
- Combination

With the individual method, you specify a value for each attribute of an output function. For example, to draw an application specific line, you specify values for the polyline attributes line type, width, and color. All subsequent calls to the output function use the attribute values you specify until you explicitly change the values.

With the bundled method, you select a set of predefined attribute values to specify the representation of an output function. VAX GKS stores predefined attribute values for each output function in a construct known as a bundle table. The bundle table consists of several sets of predefined attribute values. You specify an index into the bundle table to select a given set of attribute values for an output function. You cannot change the attribute values in the bundle tables, but you can select from several sets of predefined values to obtain the desired image representation.

With the combination method, you specify values for some attributes individually, and specify an index into a bundle table for other attributes. Thus, the combination method provides flexibility; it permits you to use predefined attribute values that meet application demands and specify only the unique attribute values.

- VAX GKS INPUT

  VAX GKS supports input functions to provide for interaction between you and the application program. The input functions enable a user at a physical input device, such as a keyboard, to supply data to an application program.

  VAX GKS accepts input from logical input devices so that the interaction is independent of the available physical input devices.

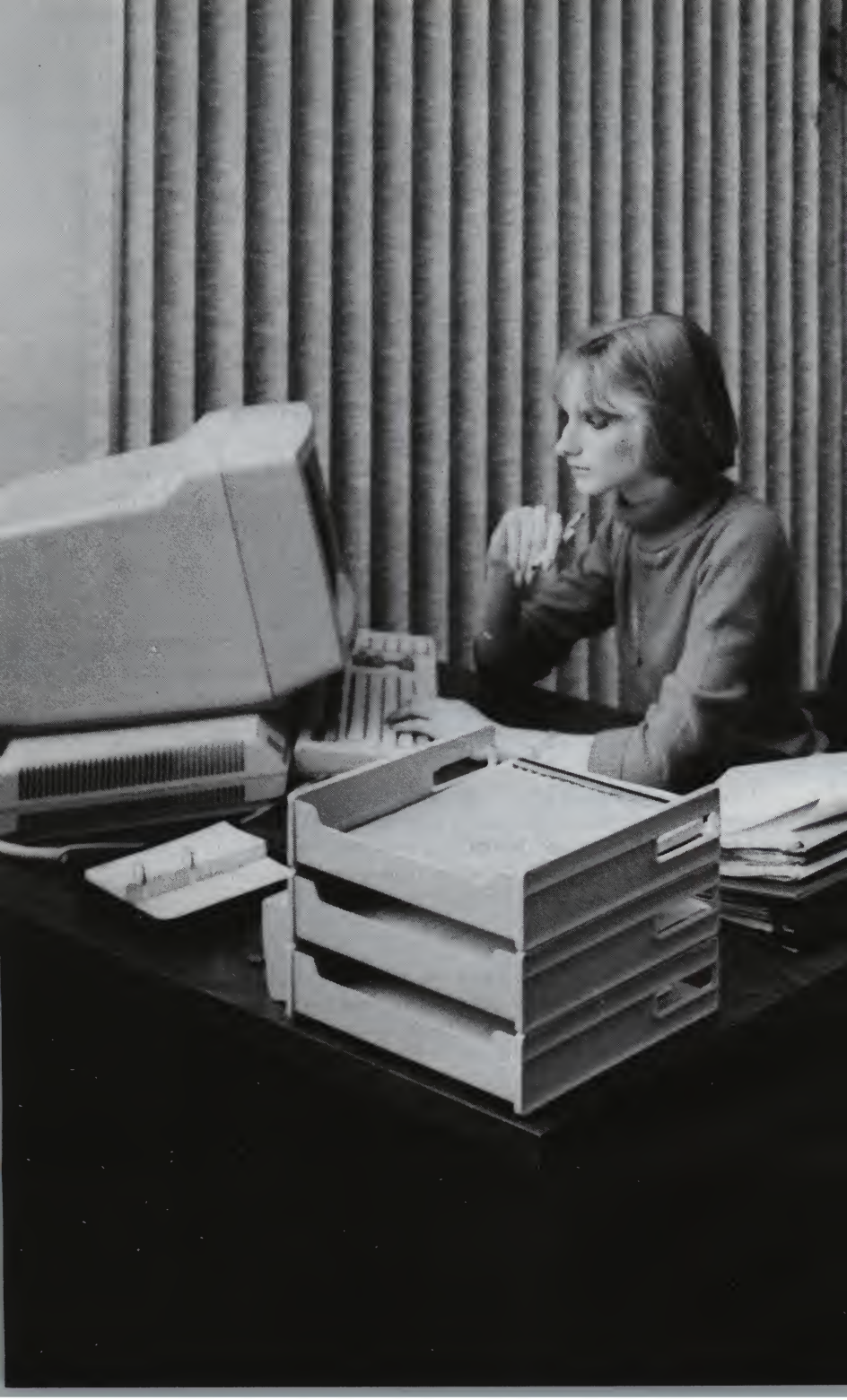
  Logical input devices are abstractions of the commonly available physical devices; they deliver logical input values to the application program. The type of data that each logical input device delivers distinguishes five basic classes of logical input devices as follows:

  - Locator — returns a World Coordinate point
  - Stroke — returns a sequence of World Coordinate points
  - Valuator — returns a real number
  - Choice — returns a selection from a number of choices (for example, a menu)
  - String — returns an individual character or an entire text string

  To accept data from a logical input device, an application program either describes the characteristics of an input device or uses initial values that provide default characteristics, and then requests input from a logical input device. Following a request for input, VAX GKS attempts to read a logical input value from the device. You control a physical input device to specify the data.

- VAX GKS METAFILES

  VAX GKS provides an interface to a system for filing graphical information. A file of graphical information is useful for external storage, exchange, and reproduction of a graphic image. The VAX GKS interface for filing graphical information is through Metafile Output and the Metafile Input.



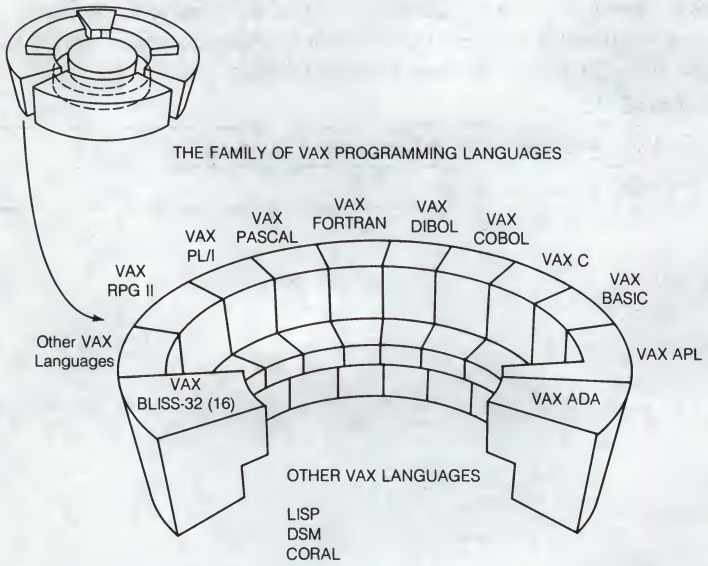
Chapter 3 ▪ VAX Programming Languages

▪ Overview

The large collection of VAX high-level programming languages is described in this chapter. Each VAX language, presented in alphabetical order, is covered in its own section that supplies information on language extensions, special features of each language, and a sample program listing.

Topics include

- The VAX Common Language Language environment.
- The VAX high-level languages.



This chapter describes the family of VAX languages that give application and systems programmers a Common Language Environment in which programs

- Can be written in more than one language that conform to a common calling standard.
- Use a common debugging utility (VMS DEBUGGER).
- Use a common runtime library (VMS RTL).
- Use a common file I/O system (VMS RMS).
- Can call operating system services and use a common handling of exceptions.

Figure 3-1 ▪ Overview of Chapter 3

▪ Introduction: General Features of VAX Programming Languages

As we saw in Chapter 1, the VAX/VMS Software Development Environment provides you with a continually evolving environment for the design of systems and applications programs that use the VMS operating system. These designs are used in a wide range of applications, including engineering, scientific, commercial, instructional, and system program implementation activities.

At the center of the productivity environment is the family of VAX programming languages. Applications and system programmers have a diversified range of higher-level and assembly-level programming languages at their disposal. In addition to the assembly-level language, MACRO, VAX programming languages cover the gamut from VAX™ Ada® to VAX RPG II.

The following table, Table 3-1, can be used to show the different components of the VAX/VMS Software Development Environment as they might be used when designing an application program that involves the use of one or more of the VAX languages. The table indicates major data types supported by the languages, support for EXTERNAL data, data in the Common Data Dictionary, and the ability to pass data and results between program modules (VAX Calling Standard support). The table indicates language support in the VAX Symbolic Debugger, access to the SORT/MERGE facilities of VMS, and whether forms management utilities can be used.

In addition, the table shows file access methods available through the VAX RMS file management system, and which access methods are supported by each of the languages. Access to database facilities is shown, including the callable interfaces to Datatrieve, DBMS (network database), and Rdb/VMS (relational data base access).

The rightmost columns of this table show integration of the Common Language Environment with the VMS Programmer Productivity Tools. These Software Tools are language-neutral in many cases, but where appropriate they do support language-specific operations, (For example, the Language Sensitive Editor (LSE.) Following the table, you will see sections describing each of the languages and the software tools in more detail.

Table 3-1: ▪ Cross-reference chart for VMS Services, VAX Languages, Software Tools, and Related Program Development Products.

	Data Support						Declare				Program Runtime				RMS File Access					Database Access			Software Tool Support				
PRODUCT NAME	INTEGER	FLOAT	DECIMAL	FIXED	VARYING	DYNAMIC	COMMON	EXTERNAL	CDD	CALL STD	DEBUGGER	SORT	FMS	TDMS	STREAM	SEQUENTIAL	BLOCK	RELATIVE	INDEXED	DATATRIEVE	DBMS	RDB	CMS	MMS	TEST/MGR	LSE	PCA
VAX Ada	y	y	y	y		y				y	y	*	*	*	*	*	*	*	*	*	*	*	y	y	y	y	y
VAX Apl	y	y				y										y		y	y	*	*	*	y	y	y		
VAX BASIC	y	y	y	y		y	y	y	y	y	y	*	*	*	*	y	y	y	y	*	*	P	y	y	y	y	y
VAX C	y	y		y	y			y	y	y	y	*	*	*	y	y	*	*	*	*	y	P	y	y	y	y	y
VAX COBOL	y	y	y	y			y	y	y	y	y	y	*	*	*	y	*	y	y	*	*	P	y	y	y	y	y
VAX CORAL	y	y		y				y		y	*	*	*	*	*	y	*	*	*	*	*	*	y	y	y	y	y
VAX DIBOL			y		y		y	y	y	y	*	*	*	*	y	y	y	y	y	*	*	*	y	y	y	y	y
VAX FORTRAN	y	y		y			y		y	y	y	*	*	*	*	y	*	y	y	*	P	P	y	y	y	y	y
VAX PASCAL	y	y		y				y	y	y	y	*	*	*	*	y	*	*	*	*	*	P	y	y	y	y	y
VAX PL/I	y	y	y	y	y		y	y	y	y	y	*	*	*	y	y	y	y	y	*	*	*	y	y	y	y	y
VAX RPG II	y	y	y	y		y	y	y		y	y	y	*	*	*	y	y	y	y	*	*	*	y	y	y		y

* means access via CALL or other language feature

y means actual Language support

P means preprocessor support

User applications can be made up of individual program modules written in many different VAX languages. These languages all conform to a single calling standard, therefore, application systems can include modules written in several languages. Modules can CALL, pass parameters, and/or read files in cooperation with modules written in other VAX languages.

▪ VAX™ Ada®

Ada is a modern, higher-order programming language designed as a result of a competition sponsored by the United States Department of Defense. Although Ada has now become the single programming language for all mission-critical Department of Defense software, it is also well suited to many civilian applications, such as CAD/CAM, or process control. Ada is ideal for large applications that must be developed and maintained by many programmers.

Features of VAX Ada

- Ada is used in a variety of applications, including systems, computational, general, and realtime programming.
- Besides providing powerful language features, Ada reduces software life cycle costs by providing for modularization and separate compilation using packages, scope rules, and a compilation database.
- Ada allows both bottom-up and top-down program development. Ada enhances software reliability through strong typing.
- In addition to being a suitable language for embedded realtime applications, Ada gives you features for multitasking, such as tasks, rendezvous, priorities, and entry calls.

Who Uses VAX Ada ?

Users of VAX Ada Include:

- Government prime contractors for which use of Ada has been specified as a mandatory requirement.
- Major industrial corporations whose work as prime contractors has led to a significant in-house Ada training effort. These internal resources are increasingly being used for non-government related, general-purpose programming.
- Educational institutions seeking, for teaching purposes, a standard programming language that demonstrates modern programming techniques.

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

VAX/VMS Implementation of Ada

The Ada Compilation System for the VMS operating system is a complete implementation of the Ada language, and conforms fully to the ANSI standard and validated by the Ada Validation Office.

The VMS Ada compilation system consists of:

- The Ada compiler
- The Ada Compilation Library Manager that provides support for programming teams through:
  - Shared use of a compilation library by many programmers.
  - The ability to share compiled Ada code either by reference or copy.
  - Use of individual libraries as sublibraries of team libraries.
  - Automatic recompilation of obsolete units.
- High-level, fully symbolic debugging capability through the VMS debugger, including support for:
  - Mixed Ada and non-Ada code
  - Packages
  - Multiprogramming
- Integration with the VMS operating system includes:
  - Conformance to the VAX Calling Standard, which provides the ability to call and be called by code written in other languages and to call VMS system services and the VMS Run-Time library.
  - The ability to handle VMS asynchronous system traps (ASTs).
  - Comprehensive diagnostic messages, including automatic syntax error correction, geared to help the new Ada user.

Ada Program Units

An Ada program, also called an Ada system, is composed of one or more units, each of which may be separately compiled. There are four types of units:

- Subprograms
- Tasks
- Packages
- Generic Units

All Ada program units have a similar two-part structure consisting of:

- A specification that identifies the calling user interface to the program unit.
- A body, which contains implementation details that can be hidden from the user.

The specification and body can be separately compiled. This aids in large development efforts by allowing all specifications to be written first and by creating a design structure. Unit bodies can be written and refined independently.

SUBPROGRAMS - Subprograms are the basic units of execution in an Ada system. A subroutine is a procedure that expresses a action.

TASKS - Tasks are independent, concurrent operations that communicate with each other by exchanging messages. Tasks are used to implement Ada's concurrent processing feature.

PACKAGES - Packages are collections of resources, such as data type declarations, data objects, subprograms, tasks, or other packages. Because portions of a package can be hidden from the user, packages are used to provide various levels of data abstraction.

The definition of the Ada programming language includes several predefined library units. Among these are:

- CALENDAR
- DIRECT_IO
- IO_EXCEPTIONS
- LOWLEVEL_IO
- SEQUENTIAL_IO
- SYSTEM
- TEXT_IO
- UNCHECKED_CONVERSION
- UNCHECKED_DEALLOCATION

GENERIC UNITS - A generic unit is a template of an algorithm that can be tailored to particular needs at compile time. A generic program unit can be created by adding a prefix to an existing program unit. This prefix is called the generic part, and it defines any generic parameters. A generic routine is called like a subroutine, and becomes nongeneric when it is called because it is passed parameters that are defined to have a certain data type.

Major Features of the Ada Programming Language

VAX Ada supports these features:

- Strong typing
- Data abstraction
- Machine-dependent facilities
- Exception handling
- Generic definitions
- Relative and absolute precision specification

STRONG TYPING - Ada is a strongly typed language. This means that an object (variable) of a given type may take on those values that are appropriate only to that type, and certain predefined operations may be performed only to data of that type. For example, an integer value cannot be assigned to a real variable without an explicit conversion.

A data type characterizes a set of values and a set of operations applicable to those values. For example, integer is a data type with values represented by any positive or negative whole number or zero. The predefined operations that can be performed on the integer data type are addition, subtraction, multiplication, division, modula, absolute value, and exponentiation.

Because type checking is done at compile time, strong typing ensures that any errors associated with incorrect data types are detected at compile time.

DATA ABSTRACTION - Data abstraction, or information hiding, obscures the details of implementation, while providing users with mechanisms for using the implementation. Abstraction allows us to focus on important characteristics while ignoring underlying details. For example, when we think of integer data, we think of whole numbers and the operations that can be performed on them, rather than the fact that integer data is actually implemented as binary 1s and 0s by the computer.

SYSTEM-DEPENDENT FACILITIES - Different systems built by various manufacturers will vary in such characteristics as the size of their storage unit, memory size, and the smallest and largest integer values supported. Ada provides a package called, System, which contains a collection of system defined constants to represent system-dependent information. The values of these constants are defined by the actual implementation.

There is also a feature called a pragma that allows the setting of parameters that describe implementation features and attributes. There are predefined pragmas, and the application is free to add others.

CONCURRENT PROCESSING - For many applications it is important that a program be conceived as a number of parallel activities rather than a serial sequence of actions.

Most high-order languages provide little or no support for handling such concurrent activities; they rely on facilities of the host operating system. Ada uses tasks to support parallel activities directly within the language.

EXCEPTION HANDLING - In many operations, especially in embedded computer systems, it is critical that a system be able to recover quickly and efficiently from error conditions.

Ada includes predefined exception conditions, and also permits the user to define exceptions. It provides the ability to raise exception conditions, and to actually handle conditions. When an exception occurs, normal processing is suspended and control passes to the exception handler.

GENERIC DEFINITIONS - In many cases, the logic of a program is independent of types of the values being manipulated. For example, in a simple exchange routine that exchanges two elements of the same data type, the algorithm to do the exchange would be the same for any data type. However, in a strongly typed language such as Ada, all types must be defined at compilation time. To use the exchange routine for various data types, a separate routine would have to be coded for each data type.

The following is an example of a typical VAX Ada program.

```
-- This program has a "background task" that sorts an integer array
-- while another task interacts with the terminal user.  The interactive
-- task, upon user command, will display the array at any time
-- during the sort.  (The QUICK_SORT procedure is not shown; it is
-- presumed that QUICK_SORT has delay statements to keep it from
-- running too fast.)

-- Package to provide I/O operations for objects of type
-- INTEGER and FLOAT
--
with TEXT_IO; use TEXT_IO;
package IOPACKAGE is
   package INTIO is new INTEGER_IO(INTEGER);
   package FLOATIO is new FLOAT_IO(FLOAT);
end IOPACKAGE;

-- Main program to sort an array and examine it as it is sorted.
-- All tasks in this program have one master, procedure TASKSORT,
-- which is also their immediate master.
--


with TEXT_IO, IOPACKAGE;
procedure TASKSORT is
   use TEXT_IO, IOPACKAGE.INTIO;

   pragma TIME_SLICE(0.3); -- enable interleaved execution


   -- Define array to be sorted.
   --


   type QUICKARRAY is array (INTEGER range <>) of INTEGER;
   A      : QUICKARRAY(1..120);
   ASIZE  : INTEGER;


   -- Specify quick sort procedure, to be provided by the user, and
   -- to be used by the sorting task
   --

   procedure QUICK_SORT(A : QUICKARRAY; ASIZE : INTEGER) is separate;

   -- Specify a single task to perform the sort
   --

   task QUICK is
      entry START (ARG: INTEGER);
   end QUICK;

   -- Specify another single task to do terminal I/O
   --
```

```
task USER is
   pragma PRIORITY(8);  -- Give "interactive task"
                        -- higher priority than sorting task
end USER;

-- Procedure to be used by the interactive task to print
-- out the array
--

procedure PRINT_ARRAY is
begin
   for I in 1..ASIZE loop
      PUT(A(I),WIDTH=>3);
   end loop;
   NEW_LINE;
end;

-- Corresponding task body for specification QUICK;
-- contains a synchronization point for starting the sort,
-- and a call to the sorting routine
--

task body QUICK is
   ARRAY_SIZE       : INTEGER;
begin
   select
      accept START (ARG: INTEGER) do
         ARRAY_SIZE := ARG;
      end START;
   or
      terminate;
   end select;
   PUT_LINE("The sorting task has started");
   QUICK_SORT(A,ARRAY_SIZE);
   PUT_LINE("The sorting task has completed");
end QUICK;

-- Corresponding task body for specification USER;
-- uses two nested loops: the outer loop allows you to
-- input the integer array for sorting; the inner loop
-- allows you to look at the array (or exit) while it is
-- being sorted
--

task body USER is
   I        : INTEGER;
   LAST     : NATURAL;
   SENTINEL : STRING(1..120) := (1..120 => ' ');
begin
   loop
      begin
         PUT_LINE("Type in the number of integers you want sorted,");
         PUT_LINE("and then type a RET.");
         GET(ASIZE);
         PUT_LINE("Now, type in a string of integers, separated by spaces,");
         PUT_LINE("that you want sorted. End the string with a RET.");
         for I in 1..ASIZE loop
            GET(A(I));
         end loop;

         PUT("The initial array is ");
         PRINT_ARRAY;

         QUICK.START(ASIZE);  -- Start the sorting task by rendezvous
                              -- with task QUICK; synchronize USER and
                              -- QUICK
```

```
            -- Allow the terminal user to see the array or exit any time
            -- he/she wants.
            --

            loop
               PUT_LINE("Type: (RET) to see partially sorted array or e to exit");
               GET_LINE(SENTINEL, LAST);

               if LAST >= 1 and then (SENTINEL(1) = 'E' or SENTINEL(1) = 'e')
               then
                  exit;
               end if;

               PRINT_ARRAY;
            end loop;
            exit;

            exception
               when END_ERROR =>
                  PUT_LINE("That's all folks!");
                     exit;
               when others =>
                  PUT_LINE("You've made a mistake; try again");
                   SKIP_LINE;
         end;
      end loop;
   end USER;

begin          -- tasks USER and QUICK are activated;
               -- environment task for TASKSORT is created.
    null;
end TASKSORT; -- procedure finishes when USER and QUICK
               -- are done; control returns to VAX/VMS
--
-- Should you run this program, you must make sure that the input
-- file is not a process-permanent file (SYS$INPUT); otherwise, the
-- lower priority sorting task will not run.  To avoid using SYS$INPUT,
-- first execute the following DCL command:
--
-- $ DEFINE ADA$INPUT TT
```

Figure 3-2 ▪ Sample Ada Program Listing

▪ VAX APL

VAX APL (A Programming Language) is a compact and versatile programming language that runs on the VAX series of computers. This language is especially suited to handle numeric and character data organized as lists and tables and is used extensively in such areas as the manipulating of data, the designing of systems, and the computing of mathematical and scientific solutions.

The language was originally designed as a notation language to explain concepts, not as a computer language. As such, APL is a language for communicating ideas, not just telling computers what to do.

Features of VAX APL

- VAX APL uses an interactive interpreter, so you don't have to compile or link programs
- VAX APL has a number of special operations that lets you handle lists and numbers as they now handle numbers.

VAX APL uses virtual memory to create a workspace that can expand as needed. Because the symbol table and execution stack (SI) are part of the workspace, they too can grow dynamically as long as memory is available.

There is also a complete set of APL system commands that can change the system environment, including listing and deleting files from disk and loading, saving, and copying VAX APL workspaces. Unlike other languages, which are compiler-oriented, VAX APL is a sharable and reentrant interpreter that allows you to edit functions or debug programs without ever leaving APL. VAX APL also provides detailed error messages to assist you in detecting and debugging errors.

Characteristics

Although it is also a high-level language like FORTRAN, COBOL, and PASCAL, APL differs from these languages in several respects.

- SIMPLE SYNTAX — Most functions in APL are designed to manipulate data. As long as the data is in an acceptable format, the manipulation will take place, allowing several functions to be placed one after another to create the APL program. The few syntax rules that do exist in APL consistently apply to both primitive (built-in) functions and the user-defined functions.
- MODULAR DESIGN — One APL program can easily call another APL program and have data returned as a result. This allows you to build a library of special purpose routines and to easily create top-down organization.
- TABLE HANDLING — APL uses very powerful primitive functions that can act just as easily on an entire table as they can on a single number. VAX APL can handle a table of up to 65535 dimensions.
- IMMEDIATE EXECUTION — VAX APL is an interactive language, therefore you can see the results of their efforts immediately after they are entered. It is often faster to use an APL function on a large table than it is to compile a loop in other languages to process the table.

To be highly productive, VAX APL users don't need to know much about the VMS operating system. The APL interpreter supplies everything that will be needed during a terminal session. In addition to providing a built-in editor, VAX APL provides debugging aids, systems communication facilities, and a file system. This means you can edit and debug a program without ever leaving APL.

Writing a program usually takes less time in APL than in many other languages. Because of this time saving characteristic, APL is also cost efficient.

In APL, you don't need to reserve space for variables (DIMENSION) or specify data types (INTEGER, REAL, etc.). Input and output format statements are unnecessary. Rows and columns of data can be manipulated without loops. Compiling and linking are unnecessary because APL can execute code immediately.

These steps are unnecessary because APL does them automatically. This saves time and allows you to concentrate on problem solving. For example a programmer can write a program to compute an average faster in APL than in FORTRAN. A FORTRAN program to average a set of unknown numbers may look like:

```
        INTEGER N
        REAL T, D

        T = 0
        N = 0

10      WRITE (*,1000)
1000    FORMAT ('$', 'Enter next number of ^Z to stop: ')
        READ (*,*,END=100) D
        N = N + 1
        T = T + D
        GO TO 10

100     IF (N .EQ. 0) THEN
          WRITE (*,2000)
2000      FORMAT (' No numbers to average')
          STOP
          ENDIF

        WRITE (*,3000) N, T/N
3000    FORMAT (' The average of the ',I4, ' numbers is ', F10.2)

        END
```

APL can use the following code to do the same thing:

```
'ENTER NUMBERS SEPARATED BY SPACES'
(+/X)÷⍴,X←⍞
```

▪ VAX BASIC

The VAX BASIC product gives you the benefits of a highly interactive programming environment and a high-performance development language. It combines the features of a compiled, structured BASIC and the RSTS/E BASIC-PLUS language with the performance benefits provided by a VAX language that is fully integrated with the VMS environment.

The VAX BASIC language is a highly extended implementation language. It provides powerful mathematical and string handling facilities, support for symbolic variable names/debugging, and full RMS indexed, sequential, and relative I/O operations.

VAX BASIC can be used as if it were either an interpreter or a compiler. A fast RUN command and support for direct execution of unnumbered statements (immediate mode) gives you the feel of an interpreter. However, this product can also be used in compilation mode, where it generates object modules like the other VAX compilers. In either case, the VAX BASIC system generates optimized VAX native-mode instructions that have extremely fast execution times.

Features of VAX BASIC

- BASIC programming support environment
- Structured programming constructs in the language
- Labels
- Conditional compilation and compile-time directives
- Alphanumeric labels on statements
- Full support for the VAX Language-Sensitive Editor and the VAX Symbolic Debugger
- Program segmentation
- User-defined data types

Who Uses VAX BASIC?

- Students, teachers, and administrators
- Third-party application development houses
- Financial institutions
- General-purpose data processing departments

General Characteristics

The VAX BASIC system generates inline VAX instructions in both its RUN and its compilation modes. The code produced takes advantage of VAX/VMS capabilities, including:

- Direct calls to operating system service routines, even in immediate mode
- Transparent use of DECnet communications software
- Direct calls to the Common Run-Time Library and standard system utilities, including VAX SORT/MERGE
- Direct calls to separately compiled native mode procedures written in any language that conforms to the VAX procedure-calling standard
- Program sizes up to 1 billion bytes are allowed
- All modules are position-independent (PIC) and can be run as fully reentrant code

The code generated by the VAX BASIC system uses the standard VMS traceback facility for determining the source of run-time errors. If a fatal program error should occur, an English message is printed identifying the module and line number where the error occurred. The English text, the traceback, and the integrated BASIC HELP utility provide a powerful program debugging environment.

Object modules produced by the VAX BASIC system can be linked with object modules produced by other language processors including the BLISS, COBOL, FORTRAN, PASCAL, and MACRO processors.

Structured Programming

Structured programming constructs add most of the features of a block structured language (such as PASCAL) to the BASIC language to allow complex programs to be written without recourse to GOSUBS or obscure programming techniques. This makes programs easier to write and maintain.

Figure 3-3 illustrates a data structure defined by the RECORD statement, successive retrievals by the use of a GET statement, and iteration controlled by a WHILE...NEXT statement block. Also, note the use of named constants and labels.

```
100     %TITLE 'VAX BASIC demo program'
        %SBTTL 'Declarations'

        OPTION TYPE = EXPLICIT                  ! Require declaration of all
                                                ! variables

        ON ERROR GOTO Error_handler
```

```
    RECORD Employee_rec                     ! What an employee's file entry
                                            ! looks like
        VARIANT
        CASE
            STRING Whole_name = 36          ! The employees whole name
        CASE
            STRING Last_name = 20           !
            STRING First_name = 12          ! Another view of his name
            STRING Middle_initials = 4      !
        END VARIANT
        DECIMAL(7,2) Rate                   ! Pay rate
    END RECORD Employee_rec

    MAP (REC) Employee_rec Employee
    DECLARE STRING File_name,                                          &
            DECIMAL(10,2) Total_rate,                                  &
            BYTE   End_flag
    DECLARE BYTE CONSTANT   True = -1,                                 &
                            False = 0
    DECLARE LONG CONSTANT   End_of_file = 11

    %PAGE
    %SBTTL 'Main code'

    File_name = 'EMPLOYEE'
    Total_rate = 0
    End_flag = False
    OPEN File_name AS FILE #1, SEQUENTIAL, ACCESS READ, MAP REC, DEFAULTNAM ".DAT"
    WHILE NOT End_flag
        GET #1
        Total_rate = Total_rate + Employee::Rate
    NEXT
    GOTO Program_end

    %PAGE
    %SBTTL 'Error_handler'
Error_handler:
    SELECT ERR
        CASE End_of_file
            End_flag = True
            RESUME
        CASE ELSE
            ON ERROR GO BACK
    END SELECT

    %PAGE
    %SBTTL 'Print result and clean up'

Program_end:
    PRINT 'Total rate: $'; Total_rate
    CLOSE #1
    END
```

Figure 3-3 ▪ Sample Structured VAX BASIC Program

The SUB and FUNCTION constructs in the VAX BASIC language have structured END and EXIT statements. In addition, this language allows the use of statement modifiers that allow conditional or repetitive execution of the statement without requiring you to construct unnecessary loops or blocks. Any nondeclarative statement in the VAX BASIC language can have one or more statement modifiers. The BASIC statement modifiers include FOR, IF, UNLESS, UNTIL, and WHILE constructs. Each of the statements in Figure 3-4 illustrates the use of a statement modifier:

```
100     A(I)=A(I)+1                 FOR I=1 TO 100

200     PRINT SUMMARY DATA          IF  OPTION.1 AND (REPORT="MONTHLY")

300     PRINT HOUSE.PAYMENT         UNTIL   RATE< 123.45

400     GET #1                      WHILE EMPLOYEE.NUMBER<76000

500     GOSUB 12300                 UNLESS ERROR.FLAG

600     PRINT"NORMAL EXIT"          IF      TOTAL>1000 UNLESS ERRORS>0
```

Figure 3-4 ▪ Statement Modifiers

▪ VAX Bliss-32

VAX Bliss-32 is a high-level systems implementation language. The Bliss-32 language supports development of modular software according to structured programming concepts by providing an advanced set of language features. It provides access to most of the hardware features of the VAX systems to facilitate programming of time-critical and hardware dependent applications.

Features of VAX Bliss-32

Bliss-32 is specifically designed for the development of:

- Many parts of operating systems
- Compilers
- Runtime system components
- Database file systems
- Communications software
- VAX utilities

What is VAX Bliss-32 Used For?

- Base operating system design and engineering
- Compiler development
- Database management systems engineering
- Cross-compiler development
- Hardware-dependent applications

The Bliss-32 compiler operating translates source programs into relocatable object modules that can be linked for execution. Bliss-32 compiled code is optimized for execution efficiency.

The following features of Bliss-32 are machine independent. Collectively, this set of features is known as "Common Bliss" and can be used to develop transportable programs that will run on VAX, DECsystem-10, DECSYSTEM-20, and PDP-11 systems.

- Modules are compiled separately for modularity and convenient development. Object modules are relocatable and can be linked with other object modules.
- The Bliss-32 language provides expressions for describing the actions to be performed and declarations for allocating, describing, and initializing data, and for defining macros and literals.
- The operators provide a set of operations for integer arithmetic, for comparison, maximization, and minimization of signed integer, unsigned integer, and address values, and for Boolean operations.
- Field references allow values to be retrieved from or assigned to any contiguous field from 1 to 32 bits located anywhere in the VAXvirtual address space.
- Character sequence functions provide for efficient runtime manipulation of character data. Operations include moving, concatenating, comparing and translating character sequences, as well as searching for particular characters or substrings of characters.

The VAX Bliss-32 Compiler

The VAX Bliss-32 compiler performs a number of optimizations. These include common subexpression elimination, removal of loop invariants, constant folding, block register allocation, peephole replacement, test instruction elimination, jump vs. branch instruction resolution, branch chaining, and cross-jumping.

The VAX Bliss-32 compiler optionally produces source text and generated code in a format closely resembling a VAX assembly listing. Other options allow you to control the degree of optimization, suppress production of object code, determine types and formats of output listings, generate traceback information, and specify the types of information to be listed at the terminal.

Library and Require Files

The Bliss-32 language provides two methods for including commonly used text into Bliss programs at compile time. These involve use of either Library files or Require files:

- Library Files — These are special files created by the compiler in a previous library compilation and are invoked by the LIBRARY declaration in the Bliss source program.
- Require Files — These are source (text) files invoked via the REQUIRE declaration in the Bliss source program.

Because Library files are precompiled, lexical processing and declaration parsing and checking need not be repeated each time these files are included in a compilation; their use results in a considerable reduction in total compilation time.

The contents of Require files must be fully processed each time the file is used in a compilation. Hence using Require files will, in general, be less efficient than using Library files. However, since these files operate under a less stringent set of syntactical rules, their use may be warranted in situations in which a higher level of flexibility is desired.

Macros

The VAX Bliss-32 language provides an extensive macro-building facility, allowing frequently used groups of declarations or expressions to be expressed in an abbreviated way. Macros are defined via MACRO declarations and are accessed by simple call statements. They are fully expanded at compile time. The Bliss-32 language allows parameters to be specified in the macro definition, thus allowing each block of text to be specialized by the actual parameters passed to it. Various types of MACRO definitions give the programmer very flexible and powerful capabilities.

Debugging

The VAX Bliss-32 compiler produces a list of error messages showing the source program line on which the error occurred followed by a description of the error. If the error is recoverable, then the compiler will generate a warning diagnostic and continue with the compilation process. If the error is serious enough to invalidate the compiler's internal representation of the module, then an error diagnostic is generated, and processing ceases following the syntax checking — no object module is produced.

Transportability Features

The VAX Bliss-32 language is designed to facilitate transportability; that is, the writing of programs that can be executed on architecturally different machines with little or no modification. The VAX Bliss-16 language, discussed later in this chapter, is a high-level implementation language for the development of systems software for use on PDP-11 systems. Several language features enhance transportability:

- The high-level language constructs may be transferred from one machine to another with little or no alteration.
- Machine-specific functions can be separated from the common, mainline code via modularization, macros, and Library and Require files.
- Machine-specific characteristics can to be passed to Bliss data structures with the use of parameters.

The following program shows how the VAX Bliss-32 language can call VAX/VMS system services and the VAX Common Run Time Procedure Library to print the current time on SYS$OUTPUT.

```
TIME_OF_DAY                                                    26-Oct-1984
15:58:34        VAX-11 Bliss-32 V4.1-746              Page   1
                                                               26-Oct-1984
15:57:08        BLISS$:[FAIMAN]EXAMPLE.BLI;4                   (1)

;    0001  0     MODULE time_of_day (main=print_time) =
;    0002  1     BEGIN
;    0003  1
;    0004  1     FORWARD ROUTINE
;    0005  1         print_time : NOVALUE;
;    0006  1
;    0007  1     LIBRARY 'SYS$LIBRARY:STARLET';
;    0008  1
;    0009  1     OWN
;    0010  1         timestr : BLOCK[23, BYTE],
;    0011  1         atimenow : VECTOR[2, LONG] INITIAL (23, timestr),
;    0012  1         timedesc : BLOCK[8, BYTE] PRESET (
;    0013  1                                         [dsc$w_length ] = 23,
;    0014  1                                         [dsc$b_dtype ] = dsc$k_dtype_
;    0015  1                                         [dsc$b_class ] = dsc$k_class_
;    0016  1                                         [dsc$a_pointer ] = timestr);
;    0017  1
;    0018  1     EXTERNAL ROUTINE
;    0019  1         lib$put_output : ADDRESSING_MODE (GENERAL);
;    0020  1
;    0021  1     ROUTINE print_time : NOVALUE =
;    0022  1     !++
;    0023  1     !    This routine calls the $ASCTIM system service to get the
date and
;    0024  1     !    time as an ASCII string, and then calls the RTL routine
LIB$PUT_OUTPUT
;    0025  1     !    to write that string to SYS$OUTPUT.
;    0026  1     !--
;    0027  1
;    0028  2         BEGIN
;    0029  2         $asctim (timbuf=atimenow);
;    0030  2         lib$put_output (timedesc)
;    0031  1         END;
```

```
                                        .TITLE  TIME_OF_DAY

                                        .PSECT  $OWN$,NOEXE,2

                              00000 TIMESTR:.BLKB   23
                              00017         .BLKB   1
                    00000017  00018 ATIMENOW:
                                        .LONG   23                      ;
                    00000000' 0001C         .ADDRESS TIMESTR            ;
                        0017  00020 TIMEDESC:
                                        .WORD   23                      ;
                      01  0E  00022         .BYTE   14, 1               ;
                    00000000' 00024         .ADDRESS TIMESTR            ;

                                        .EXTRN  LIB$PUT_OUTPUT, SYS$ASCTIM

                                          .PSECT  $CODE$,NOWRT,2

                                0000 00000 PRINT_TIME:
                                        .WORD   Save nothing            ; 0021
                                7E  7C 00002         CLRQ    -(SP)      ; 0029
                          0000' CF  9F 00004         PUSHAB  ATIMENOW   ;
                                7E  D4 00008         CLRL    -(SP)      ;
            00000000G  00       04  FB 0000A         CALLS   #4, SYS$ASCTIM  ;
TIME_OF_DAY     26-Oct-1984 15:58:34     VAX-11 Bliss-32 V4.1-746   Page 2
                26-Oct-1984 15:57:08     BLISS$:[FAIMAN]EXAMPLE.BLI;4  (1)

                          0000' CF  9F 00011         PUSHAB  TIMEDESC        ; 0030
          00000000G  00       01  FB 00015         CALLS   #1, LIB$PUT_OUTPUT  ;
                              04 0001C         RET                         ; 0031

; Routine Size:  29 bytes,    Routine Base:  $CODE$ + 0000


;   0032  1     END                             ! End of module
;   0033  0     ELUDOM




;                                   PSECT SUMMARY
;
; Name   Bytes                  Attributes
;
; $OWN$   40  NOVEC,  WRT,  RD ,NOEXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)
; $CODE$ 29  NOVEC,NOWRT,  RD ,  EXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)



;                               Library Statistics
;
;                                  ------- Symbols -------  Pages   Processing
;       File                       Total   Loaded  Percent  Mapped  Time
;
; SYS$COMMON:[SYSLIB]STARLET.L32;7  9776         9        0     581  00:01.0
```

```
;                                          COMMAND QUALIFIERS

;       BLISS/LIST EXAMPLE

; Size:          29 code + 40 data bytes
; Run Time:        00:02.4
; Elapsed Time:    00:03.0
; Lines/CPU Min:     831
; Lexemes/CPU-Min: 11394
; Memory Used:  35 pages
; Compilation Complete
```

Figure 3-5 ▪ Sample VAX Bliss-32 Program Listing

▪ VAX C

VAX C fully supports all of the language features of C, as described in *The C Programming Language** by B. Kernighan and D. Ritchie. VAX C provides program flow control constructs for logical and efficient program structuring, and a rich assortment of operators. and common run-time routines (only those UNIX routines that cannot be reasonably emulated under VAX/VMS are omitted.) VAX C even includes language extensions developed since the Kernighan and Ritchie book was published, including the structure assignment feature.

Features of VAX C

VAX C provides:

- A well-developed set of "structured" control flow operators
- Large set of mathematical and logical operators
- Data typing and conversions
- Consistent data declarations and data references
- Compiler produces very efficient optimized VAX code
- C language extensions
- Extended compiler features; listing and cross-referenced storage map, for example, which provide capabilities beyond the UNIX level
- Full VAX Language-Sensitive Editor and VAX C support

* "The C Programming Language", B. Kernighan and D. Ritchie, Prentice-Hall, Englewood Cliffs, N.J.,1978.

Who Uses VAX C?

- Those implementing systems and applications–C is a medium level language
- Those familiar with a Unix environment
- Those wishing to transport applications to and from compatible C environments

VAX C is more than just a faithful implementation of the C programming language. It is a very powerful implementation with an impressive collection of features and, as important, VAX C is an integrated VAX/VMS language product, which means that you have available all the services and program development aids that the VAX/VMS system provides.

The Language

VAX C is a versatile programming language that combines many of the features of a high-level language with the generality of MACRO, the VAX assembly language. Features include:

Program Control Flow — C uses simple, appropriate English for performing conditional loops (WHILE, FOR, DO), simple decisions (IF — ELSE), and multicase decisions (SWITCH); and for escaping loops or multicase decisions (BREAK, GOTO).

Data Types — Because C was designed to be a powerful, lean generalist among languages, it uses directly only the fundamental data types commonly represented by computers: integers of various, fixed sizes, and single and double-precision floating point numbers. VAX C also provides for user-defined, or enumerated, scalars (ENUM values) and aggregates (STRUCT or UNION). ENUM data-types are defined by writing the type name followed by an ordered list of identifiers that are the constants of that type.

Runtime Support — In order to retain its flexibility of application, the C language does not directly support many functions usually attributed to high-level languages; for example, I/O or common math routines. But most implementations of C include a common set of run-time support routines for accomplishing those tasks. VAX C includes most of the non UNIX specific runtime support offered in the Bell Laboratories version (even many of the UNIX specific routines have been emulated).

Unique To VAX C — New keywords for sharing data among program modules are offered by VAX C to augment the capability of the standard keyword for passing arguments, EXTERN. The new keywords — GLOBALDEF, GLOBALREF, and global value, which allow VAX C programs to define and reference global symbols offer an alternative method for dealing with external variables and values. They provide, in addition to enhanced data-sharing among C program modules, a convenient and efficient way for a C function to communicate with MACRO programs, with VAX/VMS system services and data structures, and with other high-level languages that support global symbol definition, such as VAX PL/I.

VAX C also provides an additional keyword, NOSHARE, that allows you to specify the shareability of data with VMS shared images built with C code.

C has an extremely powerful compiler that generates sharable, position-independent, VAX object code directly from C source programs (that is, no separate assembly step) a techniques such as excess of 3000 source statements per minute. In the process, the compiler can perform global and local optimization, for example,by doing global flow analysis, assigning automatic variables to register temporaries, and removing invariant computations from loops. The compiler also does peephole optimizations on the generated machine code. The result: VAX C produces faster and smaller programs.

The VAX C compiler will produce an annotated listing with line numbers and, optionally, an inline listing of generated machine code, expanded macros, storage allocation map, cross-reference listing of variable usage, expanded CDD records, translated "DEC/Shell" include file specifications, and compilation statistics.

Compatibility Across Implementations

Although creation of an ANSI standard for the C language is now underway, no national or international standards for the C language exist. *The C Programming Language,* however, is generally regarded as the definitive document, along with technical notices subsequently published by the American Telephone and Telegraph Company.

Certain incompatibilities among implementations do exist however, especially in machine-specific library routines. To aid creating portable programs, VAX C provides predefined constants (VAX, VMS, and VAX C).These can be used, for example, in conditional compilation to decide whether to compile source code that may not be portable. The VAX C compiler command, CC, also has an optional feature that detects several nonportable program constructions and issues warning messages.

UNIX — VAX/VMS Coexistence — The C programming language was originally developed at Bell Laboratories for creating the UNIX operating system, and it has become the language of choice for many applications developed on that system. As an aid to migrating programs from UNIX systems to VAX/VMS, the VAX C run-time library includes many of the UNIX-specific UNIX/C routines, emulated to run under VMS.

```
int value = -21

main()
{
printf(:The absolute value of %d is %d\n",value,absolute(value));
}

absolute(X)
int X;
{
if (X > 0) return X;
else return -(X);
}
```

Figure 3-6 ▪ Sample VAX C Program

▪ VAX COBOL

VAX COBOL is a high-performance implementation of COBOL. It is based on American National Standard Programming Language COBOL, X3.231974, the industry wide accepted standard for COBOL. Most features planned for the next COBOL standard, based on the specifications in the Draft Proposed Revised X3.23 American National Standard Programming Language COBOL, are also included.

VAX COBOL also supports an embedded Data Manipulation Language (DML) interface to VAX DBMS, Digital's CODASYL compliant Database Management System. Also, it allows access to common record definitions stored in the VAX Common Data Dictionary. VAX COBOL's support of features in the next ANSI COBOL standard, of the VAXInformation Architecture, and of other Digital-defined extensions to COBOL makes possible a wider range of COBOL applications on the VAX.

Features of VAX COBOL

VAX COBOL:

- Supports ANSI COBOL and VAX single and double floating point and address data types
- Supports contained and CALL statement facilities
- Supports an interface to VAX DBMS, Data Manipulation Language (DML)
- Provides for the creation of form and reports on selected terminals with the screen handling extension to the ACCEPT and DISPLAY statement
- Provides full report writing capabilities
- Provides complete sequential, relative, and indexed I/O
- Contains many features from the next proposed COBOL standard
- Full Language-Sensitive Editor Support

Who Uses VAX COBOL?

- Most widely used language for general data processing
- Banks and other financial institutions
- Many commercial applications, including:
  - Payroll
  - Accounts Receivable
  - Inventory Control

Object modules produced by the compiler can be linked with object modules produced by other VAX language processors.

Structured Programming

VAX COBOL adds some of the features of traditional structured programming languages (such as PASCAL and PL/I) to the VAX COBOL compiler. This facility makes programs easier to develop, understand, and maintain. It reduces program development and maintenance costs. The structured programming facilities supported by VAX COBOL include the EVALUATE statement, scope delimited statements, and the inline PERFORM statement.

The EVALUATE statement in a CASE -like statement is found in modern programming languages and allows the selection of statements to be executed, which are dependent on the state of program variables. Scope delimited statements simplify COBOL coding that previously required additional GO TO statements and procedure names. The inline PERFORM statement reduces program complexity by putting logic of the PERFORM inline.

The following program example illustrates the use of the structured programming facilities in VAX COBOL .

```
INITIALIZE-STATE.
.
.
.
PERFORM VARYING I FROM 1 BY 1 UNTIL I>12
        MOVE 0 TO MONTHLY-RETRIEVE-TRANSACTIONS(I)
        MOVE 0 TO MONTHLY-UPDATE-TRANSACTIONS(I)
        END-PERFORM

TRANSACTION-LOOP.
.
.
.
MOVE MONTH-INDEX TO I.
EVALUATE TRANSACTION-TYPE
    WHEN "RETRIEVE"

    WHEN "retrieve"
            READ TRANS-FILE AT END
            MOVE "EOF" TO TRANS-EOF-SWITCH
                    END-READ
            IF TRANS-EOF-SWITCH NOT = "EOF"
            THEN
                    ADD 1 TO MONTHLY-RETRIEVE-TRANSACTION (I)
                    END-IF
            WHEN "UPDATE"
            WHEN "update"
                    ADD 1 TO MONTHLY-RETRIEVE-TRANSACTIONS(I)
            WHEN OTHER
                    DISPLAY TRANSACTION-TYPE "is an invalid transaction"
                    ADD 1 TO TRANS-ERROR-CNT
                    END-EVALUATE.
            GO TO TRANSACTION-LOOP.
```

Figure 3-7 ▪ Use of VAX COBOL Structured Programming Techniques

The example illustrates the use of the inline PERFORM statement whose scope is delimited by END-PERFORM . The inline PERFORM loop initializes monthly transaction counts in preparation for the subsequent transaction analysis. The EVALUATE statement performs the transaction analysis and illustrates the typical use of this statement: a set of actions to be executed, dependent on the state of a program variable (for example, TRANSACTION-TYPE). For the cases not specifically mentioned, the WHEN OTHER imperative statement sequence is executed which, in this example, does exception reporting and a count of the transaction errors. The scope delimiters are END-PERFORM , END-READ , END-IF , and END-EVALUATE . These delimiters help to organize the program and to make the program more understandable and maintainable.

Data Types

VAX COBOL supports the data types specified in the ANSI COBOL Standard. VAX COBOL also supports, as extensions, the packed decimal (COMP 3), floating point (COMP 1), double floating point (COMP 2), and address (POINTER) data types.

The following is a summary of the data types supported by VAX COBOL :

- Numeric DISPLAY data
  - Trailing overpunch sign
  - Leading overpunch sign
  - Trailing separate sign
  - Leading separate sign
  - Unsigned
  - Numeric edited
- Numeric COMPUTATIONAL data
  - Word fixed binary
  - Longword fixed binary
  - Quadword fixed binary
- Packed decimal data (COMPUTATIONAL -3)
  - Unsigned packed decimal
  - Signed packed decimal
- Floating-point data
  - F_floating (COMPUTATIONAL -1)
  - D_floating (COMPUTATIONAL -2)
- Alphanumeric DISPLAY data
  - Alphanumeric
  - Alphabetic
  - Alphanumeric edited
- Address data
  - Pointer

▪ VAX DIBOL

VAX DIBOL (Digital Interactive Business Oriented Language) is a high-level, procedural language designed specifically for interactive data processing in the business environment. It takes full advantage of the VMS system's facilities.

VAX DIBOL is based on the DIBOL Standards Organization's DIBOL -83 definition of the language. VAX DIBOL is highly compatible with DIBOL -83 implementation on other operating systems.

VAX DIBOL consists of a DIBOL compiler, a sharable runtime library, a program debugging aid called the DIBOL Debugging Technique (DDT), and a set of utility programs that facilitate data handling, data storing, and interprogram communication.

Features of VAX DIBOL

- VAX DIBOL 's ease of use, extensive functionality, and concise syntax make it an easy to learn language, which can be used to develop commercial application programs quickly and efficiently
- VAX DIBOL 's debugging facility, DDT , makes debugging applications under development easy and straight forward
- VAX DIBOL is highly compatible across the following PDP -11 operating systems:
  - CTS 300
  - RSTS/E
  - RSX -11M -PLUS
  - MicroRSX
- Migrating DIBOL -83 programs to VAX/VMS from any PDP -11 version of DIBOL -83 requires minimal effort and expense.

Who Uses VAX DIBOL?

Users of VAX DIBOL include:

- OEM s and end users to create business applications quickly and efficiently in the following areas:
  - Manufacturing
  - Aviation
  - General business and operations for cable TV companies
  - Medical, dental, and legal billing systems
  - General business and control for retail operations
  - Accounting

VAX DIBOL provides efficient terminal handling and efficient access to the VAX Record Management Services (RMS), VMS System Services and Run-Time Library (RTL). VAX DIBOL conforms to the VAX calling standard, allowing DIBOL programs to call or be called by routines written in other languages.

DIBOL Language Statements

A DIBOL statement has one or more elements. The first element is an English-language verb that commands an action to be performed. The remaining elements are either keywords or arguments.

- Keywords modify or supplement the action of the verb.
- Arguments specify the objects of the action. They can consist of symbolic data names, references to statement labels, arithmetic expressions, or relational expressions.

DIBOL statements fall into the following functional groups:

- Compiler directive
- Compiler declaration
- Data declaration
- Data manipulation
- Control
- Interprogram communication
- Input/Output

Program Structure

A VAX DIBOL program consists of two major divisions — a data division and a procedure division. The data division contains the data declaration statements. These define data characteristics and identify of the data used by the program. The procedure division contains all of the program statements that implement the actions or tasks to be performed. The .TITLE compiler directive, for example, places headings on a DIBOL program listing to provide clearer program documentation. The PROC statement separates the data division statements from the procedure division statements. The BEGIN and END statements provide a means of "blocking" statements so that a group or block of statements can be used whenever a single statement is legal (IF-THEN-ELSE or DO-UNTIL). END can also be used at the physical end of the program, in effect forming a block of statements whose beginning is the PROC statement.

DATA TYPES - VAX DIBOL allows program data to be stored in numeric (decimal) form, such as numbers used for calculation, or in alphanumeric (alpha) form, such as names and addresses.

SUBROUTINE LIBRARIES — VAX DIBOL 's external subroutine capability allows you to develop subroutines to perform special-purpose functions without having to code the routine into each program. The subroutines can be user-developed or taken from a VAX DIBOL external subroutine library.

VAX DIBOL includes three external subroutine libraries.

- The Universal External Subroutine Library (USEL) includes subroutines that are available and that perform the same functions on all operating systems on which DIBOL -83 exists.
- The Operating System Specific Library (OSSL) includes subroutines that are available and perform similar functions on one or more of the DIBOL -83 systems.
- The Run-Time Subroutine Library (RTSL) includes subroutines that are available only in VAX DIBOL and provide functions unique to VAX .

UTILITIES - VAX DIBOL utilities include the DIBOL Debugging Technique (DDT), a Message Manage utility (DBLMSGMGR), and a status utility (DBLSTATUS).

- DDT (DIBOL Debugging Technique) — is a program debugging aid that allows you to examine and/or change program data at run-time, to set breakpoints at various places in the program, and to examine the flow of execution throughout the program.
- DBLMSGMGR (DIBOL Message Manager) — stores and retrieves messages for DIBOL programs that use the SEND and RECV statements. Messages can be sent to programs that are running concurrently with the sending program or to programs that run subsequently. The sending program designates the name of the program that is to receive the message and optionally, the terminal number on which the receiver will be running. The receiver retrieves the message by executing a RECV statement.
- DBLSTATUS (DIBOL Message Status) — allows you to examine and optionally delete any messages currently being held by the Message Manager.

Operating Procedures

After a DIBOL source program is created and edited, it is compiled by the DIBOL compiler. The DIBOL compiler generates binary operation codes that are linked into an image. They are then interpreted and executed at run time by the DIBOL RTL interpreter.

When executable programs are run, the operation codes are executed by the DIBOL run-time library interpreter, and the program output is produced.

Compilation and the DIBOL Compiler

The DIBOL compiler reads DIBOL source files and produces linkable object files.

In addition to producing object files, the compiler detects and reports syntax errors, and can produce a listing file that contains:

- A listing of the source program
- A table of variable names used in the program (symbol table)
- A table of label names used in the program (label table)
- A report of the number and type of errors that were encountered during compilation
- The error listing
- A cross-reference listing

Contents of the Listing File

The listing file consists of:

- The program listing
- The symbol table
- The label table
- The cross-reference listing

THE PROGRAM LISTING - The program listing consists of the original source code with line numbers prefixed to each DIBOL statement, with the exception of certain compiler directives, continuation lines, blank lines or comment lines.

Line numbers are used in the symbol table, the label table, and the cross-reference listing to identify the location of variable names and label names. Line numbers are also useful in debugging (DDT) and error trapping at run-time.

For each error detected during compilation, an error message appears in the line-numbered listing. Each message appears immediately after the line in which the error is detected.

THE SYMBOL TABLE - The symbol table contains descriptive information on each variable in the source code. The table consists of information on the name (NAME) of data fields used by the program, the data type (TYPE) of the field, the number of data elements in the field (DIMENSION), and the number of characters (SIZE) required to store the field.

THE LABEL TABLE - The label table contains descriptive information on each label name and external subroutine name in the source code. The table consists of the name (NAME) of each label that is used by the program and the line number (LINE) of each label definition.

THE ERROR LISTING - This report contains a list of the type and number of errors that were detected during compilation.

THE CROSS-REFERENCE LISTING - If the CROSS_REFERENCE option is selected, five additional tables are generated.

- The symbol cross-reference table
- The COMMON cross-reference table
- The label cross-reference table
- The external subroutine reference table
- The external symbol table

▪ VAX DSM

VAX DSM (Digital Standard MUMPS) is a multiuser data management and language processing system. The DSM language is a high-level, interpretive language well suited for the processing of variable-length string data. It conforms to the American National Standard MUMPS specification X11.1-1977. In addition, it provides a number of extensions.

Features of VAX DSM

- Highly interpretive, procedural language
- Requires no compilation
- DSM precompiler optimizes execution of routines
- Part of the Digital common language environment
- Provides system and library utilities, written in the DSM language

Who Uses VAX DSM?

Users of VAX DSM include:

- Medical Institutions
- Banks and Brokerage firms
- Manufacturing facilities
- Any organization that has a need for a high-performance, data management system.

Interpretive processing of the language means that each line of a DSM routine is executed as it is entered. Routines written in the DSM language do not have to be compiled or linked, making it easier to write, debug, edit, and run a job in one interactive session.

As DSM program lines are entered, the DSM interpreter examines and analyzes each DSM statement and executes the specified operation. It performs error checking during the routine execution and reports all errors at the terminal. This reduces problem-solving time, the computer time required to check the routine, and more importantly, the time required to check the routine, and most importantly, the time required to obtain a final running application.

The DSM language has many capabilities, but its basic orientation is procedural. The language is directed primarily toward the processing of variable-length string data, making interactive database systems easier to implement and maintain.

Data Management

The DSM language allows you to reference data symbolically through variables. A variable can contain either a numeric value or an alphanumeric string.

The VAX DSM system utilizes local and global variables. Local variables are defined solely for the current user (or process). Local variables are not intended for permanent storage, but only for temporary use during the life of a process.

Global variables, or simply globals, are stored on disk. Globals are referenced symbolically using names similar to those of local variables, the only difference being the circumflex preceding the first character in the variable name. Subscripted globals form a system of arrays stored on disk, the data of which forms a common database that can be made available to one or more users in the system.

Global arrays are sparse arrays; that is, the system dynamically adds nodes to the array as a user stores data in them, and deletes nodes as a user deletes them. Thus you never have to preallocate space for globals through dimensioning, nor do they have to explicitly recover disk space when they delete data.

VAX DSM provides a high performance multiway tree structure for the implementation of global variables handled by a component referred to as the Global Module. This Global Module is implemented as an integral part of the VAX DSM product. Alternatively, you can specify individual globals for storage in VAX RMS ISAM files. Specifying the default and individual globals list is done at the time the structure is created or any time the structure is not active. Subsequent data storage happens transparently at the program level. If you choose, VAX RMS ISAM can be selected as the default and individual globals specified will be directed to the Global Module.

Globals residing in VAX RMS ISAM files can be accessed by DECnet and other VAX/VMS languages. Globals maintained by the Global Module are accessible by VAX DSM only. VAX DSM provides the necessary utilities and IO language elements needed to transfer data stored in a global structure.

The Global Module provides high throughput due to caching of disk blocks.

In general, global arrays are treated syntactically in the DSM language just as local arrays. To create a global array, the SET command is issued. To access and manipulate its contents, any number of commands and functions in the DSM language set are used. To delete a global node, the KILL command is issued. To delete the entire global array, its root node is killed.

This arrangement eliminates the need to be concerned with the physical structure of files when designing a database application (as is the case with some database systems). Using globals, you need be concerned only with the logical relationship between elements of a database.

The PRECOMPILER

The VAX DSM system provides a language PRECOMPILER to optimize the execution of DSM routines in an application environment. The PRECOMPILER is a component of the DSM interpreter that processes all DSM program lines into a more efficient, intermediate format, called a precompiled format, in order to expedite subsequent interpretation.

When you store a routine, the system places both the source and precompiled versions in the DSM routine directory. For a given routine version, the precompilation procedure occurs only once. When you execute a routine from the directory, the VAX DSM system automatically loads the precompiled version.

Because the system saves both routine versions, you can always load, edit, and test DSM routines interactively. The precompilation procedure is repeated if a routine is edited or updated.

The VAX DSM PRECOMPILER transforms DSM program lines into precompiled format with the following optimizations:

- It strips comments.
- It checks syntax.
- It sets up an internal table for line labels used to optimize GOTO statements and DO statements that transfer control to other routine lines.
- It evaluates constants and transforms numbers into an internal representation (that is, packed decimal or longword).
- It converts arithmetic statements into "Reverse Polish Notation."
- It restricts the evaluation of a series of postconditionals to the occurrence of the first false condition. To do this, the PRECOMPILER generates codes that specify the appropriate offset to a given instruction.

Procedure Calls

The VAX DSM system allows you to access services that are not part of the DSM language through a Digital-implemented extension to Standard MUMPS, called the $ZCALL function. Through $ZCALL, you can call VMS system services, routines in the VAX Common Run-Time library, or routines written in other languages directly from DSM application routines. For example, the DSM language does not include a square root function. The function is in the Digital Run-Time Library, and can be called through the procedure calling mechanism.

I/O Options

The VAX DSM system provides a subset of the Input/Output (I/O) options of the VMS operating system. Each of these options can be accessed through commands in the DSM language set. You can access any VAX/VMS-supported device through commands in the DSMlanguage set.

The VAX DSM system provides an interface to VMS I/O handlers according to device type. Terminal I/O and communication through mailboxes is handled by the VMS Queue I/O service, whereas I/O to all other devices is performed through VAX RMS. This allows you to access RMS sequential, relative, and indexed files, besides, global variable files they access DECnet software.

Shared Memory Areas

The VAX DSM system supports a high degree of code and data sharing through the use of VMS memory sections. Mapping a set of precompiled DSM routines in a virtual memory section improves the performance of a DSM application because the system does not have to access DSM routines stored on disk. Instead, it can execute the routines directly from virtual memory.

VMS memory sections can be either private or shared. If shared, they are called global sections. Global sections can be created dynamically by a process or they can be permanently present in the system. Permanent global sections are generally created from routines to which a number of users require access. When a group of routines or an application is installed in a global section, all users share the same copy of precompiled DSM routines. At run time, a copy of this set of routines is mapped into the virtual address space of requesting process.

The DSM Job Controller

The DSM Job Controller is a separate process that manages interlock requests by multiple DSM user processes. It also allows system wide control over the running of DSM applications, providing functions such as enabling and disabling journaling.

Communication between a VAX DSM process and the DSM Job Controller takes place through mailboxes.

The VAX DSM system provides you with the option to use or to bypass the DSM Job Controller at DSM image activation. Work that does not affect a common database — typically program development — can bypass the job controller. Whenever you and another user are running a DSM application, interlocking requires the use of the DSM Job Controller.

Journaling

Journaling is a means of keeping a record on secondary storage (disk or magnetic tape) of transactions that alter the database (for example, global variable SETs and KILLs). VAX DSM journaling is handled by a separate process communicating with DSM users through mailboxes.

VAX DSM provides a number of journaling options to meet the needs of a system running multiple applications. Depending on the options selected, there can be one or more journal processes. One journal process can be run for each group in the system, for a number of groups in the system, or for the entire system.

System and Library Utilities

VAX DSM includes a number of utility routines written in the DSM language. These routines help you develop and maintain applications, and help the system manager control the running of DSM applications.

The utilities are divided into two categories — library and system utilities. Library utilities perform general services in three categories: procedures affecting routines, globals, and miscellaneous operations, such as numeric conversion. System utilities perform services in the areas of journaling and job control and other maintenance operations and system information.

Generally, the system and library utilities are accessed through a menu-driven utility package. Most utilities in the package are interactive. Most utilities also provide extensive online documentation that explains how to use them.

▪ VAX FORTRAN

VAX FORTRAN language specifications are based on American National Standard FORTRAN X3.9-1978 (commonly called FORTRAN-77). The VAX FORTRAN compiler supports this standard at the full-language level. Also, it provides full support for many industry-standard FORTRAN features based on FORTRAN-66, the previous ANSI standard. The qualifier /NOF77 will select the FORTRAN-66 behavior where the two standards conflict.

The VAX FORTRAN compiler:

- Produces highly optimized VAX object code
- Makes use of the VAX floating point and character string instructions
- Produces shareable code and the compiler is shareable

Features of VAX FORTRAN

- Support for all VAX RMS file formats
- Many I/O extensions
- Efficient character data handling
- INCLUDE statement
- External function and procedure calls
- Shareable programs
- Thorough optimization
- Record structure and Common Data Dictionary support

Who Uses VAX FORTRAN?

- Scientific users
- Technical users
- Educational users
- Realtime application writers

Language Characteristics

CALLING EXTERNAL FUNCTIONS AND PROCEDURES - FORTRAN programs can call subroutines and functions written in other VAX native-mode languages and system services. Special operators exist for passing argument values directly, by reference, or by descriptor. A special operator also exists for obtaining the argument values used by the system services procedures.

SHARED PROGRAMS - The FORTRAN language can be used to create shareable images, which can be used by any program written in a native programming language.

DIAGNOSTIC MESSAGES - Diagnostic messages are generated when an error or potential error is detected. Errors detected during compilation are reported by the compiler, and include source program errors, such as misspelled variable names, and missing punctuation marks.

Source program diagnostic messages are classified according to severity: F (Fatal), E (Error), or W (Warning). F-class messages indicate errors that must be corrected before compilation can be completed. Object code is not produced. E-class messages indicate that an error was detected that is likely to produce incorrect results; however, an object file is generated. W-class messages are produced when the compiler detects acceptable but nonstandard syntax; or when it corrects a syntactically incorrect statement. The message indicates the existence of possible trouble in executing the program.

The VAX FORTRAN compiler optionally produces diagnostic messages for VAX FORTRAN extensions to ANSI FORTRAN-77. This flagger can check both syntax and source form extensions.

DEBUGGING FORTRAN PROGRAMS - The VAX FORTRAN language provides facilities to aid the debugging of programs written in native mode. It accomplishes this via a program known as the interactive symbolic debugger. The debugger can be linked with a native program image to control image execution during development. It can be used interactively or can be controlled from a command procedure file. The debugging language is similar to the VMS command language. Expressions and data references are similar to those of the source language used to create the image being debugged.

Debugging commands include the ability to start and interrupt program execution; to step through instruction sequences; to display source statements; to call routines, to set break or trace points; to set default modes; to define symbols; and to deposit, examine, or evaluate virtual memory locations.

COMPILER OPERATIONS AND OPTIMIZATIONS - The VAX FORTRAN compiler accepts sources written in the FORTRAN language and produces an object file which must be linked prior to execution. The compiler generates VAX native machine language code. It will also generate an optional cross-reference listing.

During compilation, the compiler performs many code optimizations. The optimizations are designed to produce an object program that executes faster than an equivalent nonoptimized program. Also, the optimizations are designed to reduce the size of the object program.

The VAX FORTRAN compiler performs the following optimizations:

- Constant folding — constant expressions are evaluated at compile-time.
- Compile-time constant conversion.
- Compile-time evaluation of constant subscript expressions in array calculations.
- Constant pooling — only a single copy of a constant is allocated storage in the compiled program. Constants that can be used as immediate mode operands are not allocated storage. For example, logical, integer, and small floating-point constants are generated as immediate mode or short literal operands wherever possible.
- In-line expansion of statement functions.
- Argument list merging — If two function or subroutine references have the same arguments, a single copy of the argument list is generated.
- Branch instruction optimizations for arithmetic logical IF statements.
- Elimination of unreachable code — An optional warning message is issued to mark unreachable statements in the source program listing.
- Recognition and replacement of common subexpressions.
- Removal of invariant computations from loops.
- Local register assignment — Frequently referenced variables are retained (when possible) in registers to reduce the number of load and store instructions.
- Assignment of frequently used variables and expressions to registers across loops.
- Reordering expression evaluation to minimize the amount of temporary registers required.
- Delaying negation/not to eliminate unary complement operations.
- Flow-Boolean optimizations.
- Strength Reduction — Multiply operations used in array indexing are reduced to adds. Efficient Auto Increment and Auto Decrement address modes are used wherever possible.
- Jump/Branch instruction resolution — The Branch instruction is used wherever possible to eliminate unnecessary Jump instructions. This reduces code size.
- Peephole optimizations — The code is examined on an operation-by-operation basis to replace sequences of operations with shorter and faster equivalent operations.

When debugging FORTRAN programs, you can disable optimizations that would remove not-referenced statement labels, FORMAT statement labels, and immediately referenced labels. This ensures that all statement labels are available to the debugger.

▪ VAX LISP

For over 20 years, LISP has been one of the fundamental tools used in Artificial Intelligence (AI) research. The LISP ("LIS"t "P"rocessing) programming language is based on a paper, published in 1958 by John McCarthy, dealing with non-numeric computation. It differs from the majority of higher-level programming languages in that LISP programs do not use numeric computation as a basis for program execution, (although it does support facilities for numeric computation).

LISP is particularly useful for the manipulation of symbolic data. Symbols can be thought of as words; lists of symbols are then equivalent to sentences or statements. Because LISP's symbolic processing and knowledge representation capabilities can be easily used to represent human thought patterns and associations, the LISP programming language has become an essential tool for Artificial Intelligence researchers as they attempt to make computers simulate human behavior and thought.

LISP is also a great general-purpose language used for in a wide variety of applications. Any imaginable type of program be written in LISP; entire operating systems have been written in LISP.

Features of VAX LISP

VAX LISP, an implementation of Common LISP, is an extremely important dialect of LISP. VAX LISP provides you with:

- Common LISP compliance
- Rich set of data types and functions
- VMS integration
- Fully interactive interpreter
- Compiler
- Dynamic linking
- Lexically scoped variables
- LISP-sensitive editor
- Program debugging facilities
- LISP program-formatting utility

- COMMON LISP COMPLIANCE
  Common LISP is rapidly becoming the de facto standard for the LISP programming language. Major corporations and government agencies are moving quickly to standardize on this dialect.

- VMS INTEGRATION
  VAX LISP can be integrated with VMS. There is no need for you to abandon or convert software written in other VAX languages. Programs written in VAX LISP can call non-LISP routines that conform to the VMS calling standard. LISP programs can access RMS, DATATRIEVE, VAX DBMS and VAX Rdb/VMS, VMS utilities and VMS system services. It can also access your software that conforms to the calling standard.

- LEXICALLY SCOPED VARIABLES
  The VAX LISP compiler and the interpreter both generate code which executes identically. Earlier implementations of LISP frequently included compilers whose semantics differed from those of the interpreter.

- LISP EDITOR
  VAX LISP provides its own language-sensitive editor with multiple window capabilities. Since the editor is written in LISP, it is readily extensible. You can define new editor functions in Lisp and bind them to keyboard keys.

- DEBUGGING FACILITIES
  A full set of LISP debugging utilities are included. These utilities are integrated by a common command interface. The LISP debugger allows, for example, examination of the state of a running program, including the stack, variables and functions. A stepper allows you to step through the execution of a program, one line at a time. The trace utility allows the automatic display of function calls and returns during program execution.

Who Uses VAX LISP?

VAX LISP is used in the artificial-intelligence market place by:

- The research community — Investigators of human intelligence
- OEM's and software houses — Creating artificial intelligence applications for sale to end users
- Industry and government — Using the technology for internal applications

Using VAX LISP

This section provides a general introduction to the use of VAX LISP. The following topics are covered:

- Invoking LISP
- Using command levels
- Controlling input
- Creating programs

- INVOKING LISP

You invoke an interactive VAX LISP session by typing the DCL command LISP. When it is executed, a message identifying the VAX LISP system appears and the LISP prompt is displayed. To exit, enter (EXIT) and you will be returned to the DCL command level.

- USING COMMAND LEVELS

VAX LISP gives you various time saving and ease-of-use facilities. One of the most important of these, the top level read-eval-print loop, provides the basic means to write and execute programs. In addition, VAX LISP also offers a break loop, and debugger and stepper facilities. When any of these facilities is invoked by means of a function call, an error, or some other event, it establishes a "command level." A command level represents a point of interaction between you and the program and is assigned a number. The highest numbered levels represent the current level of interaction between you and the program, while the lower-numbered levels represent interactions that have been temporarily suspended. Nothing prevents the same facility from being invoked more than once. There can be multiple command levels representing the same facility. For example:

```
Lisp> (break)

break 1> (+ *counter* 1)

Fatal error in function SYSTEM::%EVAL (Signaled with ERROR).
Symbol has no value: *COUNTER*

Control Stack Debugger
Frame #7: (EVAL (describe *counter*))
DEBUG 2> (BREAK)
BREAK 3> (describe *COUNTER*)
```

```
Fatal error in function SYSTEM::%EVAL (signaled with ERROR).
Symbol has no value: *COUNTER*

Control Stack Debugger

Frame #19: (EVAL (describe *counter*))
Debug 4> ....
```

In this example, you invoke a break loop and make an attempt to use the special variable *counter*, which has no value, causing the debugger to be invoked. Then you can invoke another break loop and accidentally make the same mistake again, causing another debugger level to be invoked.

- CONTROLLING INPUT

  When using VAX LISP, expressions are entered one line at a time. Once you move to a new line, you cannot go back to the previous line. However, you can recover an input expression or an output value by using one of ten unique variables (see *Common LISP: THE LANGUAGE* for a detailed description of each).

  The following example illustrates the use of the plus sign (+) variable that is bound to the expression most recently evaluated:

```
Lisp> (cdr '(a b c)
(B C)
Lisp> +
(CDR (QUOTE (ABC)))
Lisp>
```

  You can use the <DELETE> key and several control characters on your terminal keyboard to control input. The <DELETE> key allows you to delete characters that are to the left of the cursor on the current line of input.

- CREATING PROGRAMS

  The most common way of creating a LISP program is as a source file with a text editor. The program is loaded into the LISP environment by means of the LISP LOAD function.

Although you can compose source programs with any text editor, the VAX LISP Editor provides facilities that help you enter and edit LISP source code. For example, the editor helps you balance parentheses and maintain proper indentation. Furthermore, this editor, being integrated into the LISP environment, can be extended to fit your personal editing style, and can also be used to evaluate code without leaving the editor. You can go back and forth between the editor and the read-eval-print loop at will.

Another way to create LISP programs is to define them using the interpreter in an interactive LISP session. If you define functions with DEFUN and macros with DEFMACRO, the definitions immediately become part of the interpreted LISP environment. You can then invoke your defined functions and macros. However, since these definitions are not in a permanent text file, your work is stored only temporarily and disappears when you exit VAX LISP, unless you use the editor to write the function definition out to a file. Entering programs by means of the interpreter is most useful for experimenting with small functions and macros.

The following definition of the FACTORIAL function is an example of a LISP program. It can be written in the following format in a file or in an interactive LISP session:

```
        (defun factorial (n)
(if (<= n 1) 1
        (* n (factorial (- n 1)))))
```

(defun) indicates this example is a function definition. (factorial) is the name of the function. (n) is the argument list; that is, (factorial) takes one argument, n. When (factorial) is called, the code following the argument list is evaluated and the last result computed is returned as the value of the function.

▪ VAX PASCAL

VAX PASCAL is a multipass, optimizing compiler that is a powerful superset of the PASCAL language as defined by Jensen and Wirth in PASCAL User Manual and Report (1974). VAX PASCAL accepts programs that are compatible with either the ANSI/IEEE 770X3.97 standard or the ISO standard (DIS 7185).

PASCAL's block structured nature, flexible data types and English-like statements result in significant ease-of-use benefits. These benefits include ease of program generation and ease of reading, modifying, and maintaining programs. VAX PASCAL offers the incremental capability of creating a productivity environment in which many programmers can work simultaneously, and relatively independently, on the same project.

Features of VAX PASCAL

Standard PASCAL provides a modular, systematic approach to computerized problem solving. Major features of the language are:

- INTEGER, REAL, CHAR, BOOLEAN, enumerated, and subrange data types
- User-defined data types
- ARRAY, RECORD, SET, and FILE structured data types
- Constant identifier definition
- FOR, REPEAT, and WHILE loop control statements
- CASE and IF-THEN-ELSE conditional statements
- BEGIN...END compound statement
- GOTO statement
- GET, PUT, READ, WRITE, READLN, and WRITELN I/O procedures

Who Uses VAX PASCAL

Users of VAX PASCAL include:

- Educational institutions for training
- Industry
- System's engineers
- Writers of business applications

The VAX PASCAL language takes advantage of the VAX hardware floating point, character instruction sets, and virtual memory capabilities of the VMS operating system. Features common to other languages of the VMS operating system are available through the VAX PASCAL language including:

- VAX symbolic debugger support
- Separate compilation of modules
- Standard call interface to routines written in other languages
- Access to VMS system services
- Access to all RMS file organizations
- Access to CDD data declarations
- VAX Language-Sensitive Editor Support

At compile time, options available to the process include:

- Run-time checks for illegal assignment to set and subrange variables, illegal array and string subscripts, illegal case selectors, integer overflow, and illegal pointers
- Cross-reference listing of identifiers
- Source program listing
- Machine code listing

In addition to the features mentioned above, the VAX PASCAL language incorporates the following extensions to standard PASCAL, some of which are common in PASCAL implementations:

- Lexical
  - Uppercase and lowercase letters are treated identically except in character and string constants
  - New reserved words: MODULE, OTHERWISE, VALUE, REM, a%REF, %DESCR, %IMMED, %INCLUDE, and %STDESCR
  - The exponentiation operator, **
  - Concatenation operator +
  - Hexadecimal and octal constants
  - DOUBLE constants
  - $ and (underscore) characters in identifiers

- Predefined data types
  - DOUBLE
  - SINGLE
  - QUADRUPLE
  - VARYING character strings
  - UNSIGNED
- Predeclared routines
  - I/O (OPEN, CLOSE, RESET, REWRITE, EOF, EOW, STATUS,...)
  - Arithmetic (ABS, SQR, SIN, COS,...)
  - Ordinal (PRED, SUCC)
  - Boolean (ODD,UNDEFINED)
  - Transfer (CHR,DBLE,INT,ORD,...)
  - Dynamic allocation (ADDRESS, NEW, DISPOSE)
  - Character string manipulation (INDEX, LENGTH, SUBSTR, PAD,...)
  - Unsigned (UAND, UNOT, UOR, UXOR)
  - Allocation size (SIZE,NEXT,BITSIZE,BITNEXT)
  - Miscellaneous (CARD, CLOCK, DATE, EXPO, HALT,...)
- Other extensions
  - READ (or READLN) of user-defined ordinal type
  - READ (or READLN) of string
  - WRITE (or WRITELN) of user-defined scalar type
  - WRITE (or WRITELN) of any data using binary hexadecimal or octal format
  - %INCLUDE directive
  - VALUE initialization
  - OTHERWISE clause in CASE statement
  - External procedure and function declarations
  - Conformant array parameters
  - Optional attribute specification on types, variables, routines, and compilation units

- Separate compilation
  - A MODULE capability for combining procedures, functions, and other declarations for compilation apart from the main program
  - ENVIRONMENT and INHERIT attributes to control separate and independent compilation
  - External variable, procedure and function declarations

The OPEN, CLOSE and FIND procedures extend the I/O capabilities of the PASCAL language. The OPEN procedure can contain file attributes that define the creation or subsequent processing of the file. A FIND procedure is another extension to the language for direct access to sequential files of fixed length records. The standard I/O procedures of GET, PUT, READ, WRITE, READLN and WRITELN are also available in the VAX PASCAL language.

The extended parameter specifications %DESCR, %IMMED, and %STDESCR are added to the Pascal language to denote the method of argument passing when calling a system service, procedure, or function in other VAX Languages.

```
[IDENT ('1-01')] PROGRAM Setddir (OUTPUT);

(*  This program calls the RMS procedure $SETDDIR to    *)
(*  change the default directory for the process.       *)

TYPE
    Word__Integer = [WORD] 0..65535;

VAR
    Dir__Status : INTEGER;

FUNCTION SYS$SETDDIR (
    New_Dir     : [CLASS_S] PACKED ARRAY [1..u:INTEGER] OF CHAR;
    Old_Dir_Len : Word_Integer := %IMMED 0;
    Old_Dir     : VARYING [lim2] OF CHAR := %IMMED 0)
        : INTEGER; EXTERN;

BEGIN  (* main program *)

    Dir_Status := SYS$SETDDIR ('[COURSE.PROG.PAS]');

    IF NOT ODD (Dir_Status)
    THEN
WRITELN ('Error in SYS$SETDDIR call.');

END.  (* main program *)
```

Figure 3-11 ▪ Sample VAX PASCAL Program Listing

▪ VAX PL/I

VAX PL/I is an extended implementation of the General Purpose Subset (X3.74-1981, "Subset G") of ANSI PL/I, X3.53-1976. PL/I was designed to be useful in scientific, commercial, and system programming, especially on small and medium-sized computer systems. The goals of the design of Subset G were to include features that are easy to learn, easily portable from one computer system to another, and to exclude seldom used features that increase runtime complexity.

Features of VAX PL/I

VAX PL/I provides:

- The VAX PL/I compiletime preprocessor allows language extension and conditional compilation.
- Control constructs, including DO loops, IF-THEN ELSE, BEGIN-END, LEAVE, SELECT-WHEN-OTHERWISE, and CALL statements add power to program control.
- Full PL/I features include AUTOMATIC initializations, AREA (user allocation), OFFSET, scalar assignment to arrays, the REFER structure, the ENTRY statement, and the LIKE attribute.
- A full complement of VAX data types.
- Block structuring of code reduces the cost of program development and support.
- Access to the Common Data Dictionary.
- Full Support for the VAX Symbolic Debugger and Language-Sensitive Editor.

Who Uses VAX PL/I

- The general-purpose market — has a wide range of features that appeal to a general audience.
- Commercial applications writers — because of its data manipulation and structuring capabilities; an alternative to COBOL.
- Educational market — frequently taught at the university level.
- Implementation language — for commercial and scientific applications.
- System programming — current standards enforce portability.

VAX extensions to Subset G provide additional language features that allows you to take advantage of the facilities of the VAX/VMS operating system and its components.

Extensions provided in the VAX PL/I language include selected features of the full PL/I language that were excluded from subset because of their implementation cost on computers with restricted memory and/or address space.

You can either restrict their programs to Subset G — compatibility with other implementations of the Subset G — or they can take advantage of the full PL/I features and VAX extensions in programming applications.

Applications

DATA PROCESSING - Data processing applications can take advantage of the extensive character handling functions and data structuring capabilities of the PL/I language. By declaring variables within a structure, the program can easily refer to entire records or to fields within records by referencing the name of the structure or the name of a variable within it.

In addition, the VAX PL/I language provides extended access to the features of VAX Record Management Services (RMS). By specification of ENVIRONMENT options or special options supplied for input/output statements, PL/I programs can dynamically specify RMS optimization parameters and values, spool a file to a printer or batch job queue, and set or change the protection on a file.

The VAX PL/I language supports all RMS (Record Management Services) file organizations, including sequential, relative, and indexed sequential. It also permits block-mode input/output operations. Using PL/I statements, a program can read, write, delete, and update records. Using built-in file handling functions provided by the VAX PL/I language, a program can call RMS file handling services to forward space or backward space a file or volume, to increase the allocation of a disk file, or to obtain information about the properties of a file.

VAX PL/I also supports the VAX Common Data Dictionary, allowing record descriptions to be stored in and retrieved from the CDD.

SCIENTIFIC - Scientific applications can use the PL/Iarray-handling capabilities to define arrays of up to eight dimensions. Common arithmetic and trigonometric functions are defined within the language. The VAX PL/I language supports all of the VAX hardware's floating-point data types.

SYSTEM PROGRAMMING - System programming applications can use PL/I language features to allocate storage dynamically, process linked lists and queues, and perform a wide range of bit string functions and operations.

In addition, VAX extensions to the language provide a simple means to refer to VMS system global symbols and data structures. VAX PL/I programs can take advantage of the VAX linker's allocation of storage by defining variables either as read-only or as global symbols.

Full access to all of the VAX/VMS operating system's services and procedures is possible through VAX PL/I extensions to support the VAX Calling Standard. Procedures written in the PL/Ilanguage can call and be called by procedures written in any other native mode language.

Error and Condition Handling

VAX PL/I compiler generates information in the object module of a PL/I procedure, so that when an error occurs at run time, the VAX condition handling facility can report the error and provide a module traceback.

Within the PL/I language, extensive condition handling capabilities are available via the ON statement, which allows a program to define the action to take in the event of hardware arithmetic exceptions and errors that occur during file processing.

VAX extensions to the ON statement permit the specification of condition handlers for any specific hardware or software condition that can occur.

Debugging Facilities

The PL/I compiler generates useful diagnostics that signal syntactical errors and language violations. Most compiler messages are two or three lines long and explain on how to correct the indicated error.

The VAX Symbolic Debugger supports symbolic, source line debugging of PL/I programs. You can set breakpoints in PL/I programs, examine and change variables, examine program source code, and monitor the calls and function references that occur.

Libraries

The VAX PL/I language is fully compatible with the VAXRuntime Library and provides additional runtime procedures for language support.

Source file library support is provided by the %INCLUDE statement, which allows a program, specified at compile time, an external file from which source statements are to be read. Included files can also be collected in VMS text file libraries. The VAX PL/I compiler searches specified libraries for the names of the modules included.

Performance

The VAX PL/I compiler is a sharable, VMS image that can be run on any supported VAX/VMS configuration. It produces optimized, sharable, VMS object code that is runtime compatible with other VAX languages.

You control the degree of optimization performed by the compiler at compile time, by qualifiers on the PL/I command.

The VAX PL/I Runtime Library is a sharable image, allowing for efficient linking of PL/I programs and better utilization of system resources.

Programming Example

The following program example obtains data about state flowers from a data file, STATEDATA.DAT and makes use of an INCLUDE file, STATE.TXT. The notes keyed to the sample program appear in the figure below.

```
/* METRIC CONVERSION PROGRAM */
CONVERT: PROCEDURE;
DECLARE (INVALUE,OUTVALUE) FIXED DECIMAL (10,2)
        (INUNIT,OUTUNIT) CHARACTER(2);
    DECLARE UNITS (6,2) CHARACTER (2) STATIC INTERNAL <<1>>
        INITIAL ( 'in','cm','cm','in','ft','m '
        'ft','mi','km','Km',mi'
DECLARE FACTORS (6) FIXED DECIMAL (10,2) STATIC <<2>>
    INITIAL (2,54,0,39,0,30,3,28,1,61,0,62);
DECLARE INDEX FIXED BINARY;

    ON ENDPAGE (SYSPRINT);

    INDEX = 1;
    DO WHILE (INDEX  ^=0);
        PUT SKIP LIST ('Enter conversion mode:');
        PUT SKIP;
        PUT SKIP LIST  ('1 - inches to centimeters');
        PUT SKIP LIST  ('2 - centimeters to inches');
        PUT SKIP LIST  ('3 - feet to meters');
        PUT SKIP LIST  ('4 - meters to feet');
        PUT SKIP LIST  ('5 - miles to kilometers');
        PUT SKIP LIST  ('6 - kilometers to miles');
        PUT SKIP LIST  ('0 - exit');
        PUT SKIP;
        PUT SKIP LIST ('mode 3  ');
        GET LIST (INDEX); <<3>>
        IF INDEX = 0 THEN DO;
            IF (INDEX > 0) & INDEX < 7) THEN DO; <<4>>
                PUT SKIP LIST ('Enter value to convert:  );
                GET LIST (INVALUE);
                OUTVALUE = INVALUE * FACTORS(INDEX); <<5>>
                INUNIT = UNITS(INDEX,1);
                OUTUNIT = UNITS(INDEX,2);
                PUT SKIP LIST (INVALUE, INUNIT,
                                ' = ' OUTVALUE, OUTUNIT) <<6>>
                END;
                ELSE PUT SKIP LIST( 'Invalid code - retry');
            END;
        RETURN;
        END;
```

<<1>> Unit stores the abbreviation for the measurement unit.

<<2>> Factors stores the conversion factor that will be used to convert from each specified unit to the desired unit.

<<3>> Read in the conversion code.

<<4>> Check to see that a value code was specified.

<<5>> Perform the conversion.

<<6>> Print out the results.

Figure 3-12 ▪ A sample PL/I Program Listing

▪ VAX RPG II

RPG (Report Program Generator) is a powerful, business-oriented language specifically oriented toward generating a wide variety of simple and complex business reports. RPG is a partially nonprocedural language, and is therefore not suited to all business applications. However, where it's appropriate, RPG can significantly increase your productivity and greatly improve turn-around time for generation and file maintenance application development cycles.

RPG II is an enhanced version of RPG, which was developed by International Business Machines Corporation in the early 1960s. RPG IIincorporates a wide variety of additional features not present in the original RPG, and provides extra advantages in simplicity, ease-of-use, and power. RPG II has become a popular and widely used business application language. It is the primary language for many small business users.

The VAX implementation of RPG II has been extended and enhanced to operate within the context of the VAX architecture, and to take advantage of the special features and capabilities of that architecture.

Features of VAX RPG II

VAX RPG II provides:

- Support of industry standard RPG II specifications
- CALL extensions on the calculation specification
- VAX RPG II editor is tailored to the language structure
- Integration into the VAX Common Language Environment Including:
  - Use of RMS for file processing (including sequential, relative,indexed (single and multikey) files)
  - Full integration with the VAX DEBUGGER
- Automatic record matching and merging operations for multifile processing
- Multilevel control break handling
- Record identification codes
- Table and array processing
- Field editing features
- Compile and runtime performance comparable to VAX COBOL
- Can call other languages that conform to the VAX calling standard

Who Uses VAX RPG II?

Users of VAX RPG II include:

- Software houses for commercial application development
- Newspaper/publishing for generation of mailing labels
- Data processing departments in large organizations for report generation
- Small businesses for multipurpose computing
- Health care industry for administrative purposes

VAX RPG II runs under the VMS and MicroVMS operating systems. A compiler and editor are integral parts of the RPG II package. VAX RPG II runtime support is provided with the base operating system. Therefore, VAX RPG II programs can run on any VMS or MicroVMS operating system.

The VAX RPG II compiler, using an RPG II source program as input, produces an object module. That module is input to the VAX Linker, which produces an executable image. The VAX RPG II compiler optionally produces these development and debugging aids:

- Source listing with embedded diagnostics indicating the line and column of any source code error
- Machine language code listings
- Cross-reference listing

Language Characteristics and Functions

RPG II is a language processor that accepts and interprets seven different types of fixed-format specifications. RPG II is a partially nonprocedural language; all specification types, except calculation specifications, describe data to be processed rather than processing steps. For this reason, RPG II is best suited to handle applications that require relatively simple field processing. RPG II also provides a comprehensive set of sophisticated functions best utilized for these straight-forward applications.

The general rule for determining whether RPG II should be considered as the development language for an application is quite simple: the less complicated the field handling required in the application, the more suitable RPG II is for that application. RPG II greatly enhances your productivity in such cases.

RPG II is easy to learn and use because little program logic is required. A single source statement completely defines a file's structure; similarly, input and output statements require only a few lines of code. Output records are always made up of previously defined or edited fields or constants; you need only specify the desired field name in the output specification, and RPG II performs all necessary field transfer automatically.

A VAX RPG II program is composed of a set of user-defined specifications of input data, output formats, and necessary calculation steps. You can define seven types of specifications; each is listed and briefly described below.

- Control specification — Defines control information such as collating sequence, forms positioning information, decimal separator character and currency character.

- File specification — Identifies data file parameters including file name, record size, file organization, and access mode. VAX RPG II supports sequential, direct, and indexed file organization through the VAX Record Management Services (RMS). VAX RPG II enhances the flexibility of standard RPG II by not requiring a primary file in every program. Using RMS, VAX RPG II provides file sharing with automatic record locking for relative and indexed files.

- Extension specification — Provides descriptions of tables, arrays, and record address files.

- Line counter specification — Specifies the number of available print lines for printer output files, thus defining page size.

- Input specification — Identifies record types and other control information relevant to input data files; specifies field location, field names, data formats, and control-level information for individual data fields in an input file. VAX RPG II supports data files with these formats:
  - Alphanumeric
  - Overpunched numeric
  - Packed decimal numeric
  - Word-binary numeric
  - Longword-binary numeric

- Calculation specification — Describes the specific calculation operations to be performed on the data, and specifies the order in which they are to be performed. Calculation specifications provide RPG II with its only truly procedural component. VAX RPG II supports almost all standard RPG II operation codes, and also provides several other operation codes for calling routines written in other languages, for obtaining the services provided by a the Run-Time Library, and for invoking VMS system services.

- Output specification — Describes the formats of output files and printed reports, including fields, output field editing operation, forms spacing, and constant information (such as report titles and headings). VAX RPG II extends services normally provided by RPG IIwith the delete option, which allows a program to delete records from direct and indexed files.

The VAX RPG II Editor

The VAX RPG II Editor is a full-screen, keypad editor specifically tailored to the development and maintenance of RPG II code. The following features of the editor make it a particularly valuable tool.

- Overstrike mode keeps entries in their proper columns; this is important feature because RPG II is position oriented
- A ruler displayed along the top of the editing area helps you locate and keep track of column positions and field locations
- The editor automatically sets tabs according to the type of specification being edited
- Extensive on-line help is provided

The VAX RPG II Editor's help facility is a particularly valuable productivity tool. The editor supports a dual-window layout in which the edited code appears on the lower half of the screen, and various types of help information appear on the top half. Help information may consist of:

- A keypad diagram, showing the location of editing functions keys
- Help information for a particular editing key or editing function
- Column headings for the particular RPG II specification being edited — these will be changed automatically when you begin to create or edit a different type of specification. You invoke any of the help functions by simply pressing the HELP key.

  Other productivity features provided by the VAX RPG II Editor include string-searching capabilities.
- The VAX RPG II editor also provides the COMPILE command that allows you to compile programs being edited and to review and correct compilation errors without ever leaving the editing environment.

```
FIN      IP  F    1280            TAPE   TAPE                    N
FOUT     O   F      80            DISK
E                    DATA      16 80
IIN
I                                              11280 DATA2
C            TRANS      EXTRN'LIB$TRA_EBC_ASC'
C                       CALL TRANS
C                       PARMD          DATA2
C                       PARMD          DATA2
C                       MOVEADATA2     DATA
C                       Z-ADD1         I       20
C            LOOP       TAG
C                       EXCPT
C            1          ADD  I         I
C            I          COMP 16                      9999
C   99                  GOTO LOOP
OOUT     E
O                           DATA,I  0080
```

Figure 3-13 ▪ Typical RPG II Program Showing a CALL Statement

```
FRPGVEN  IP  F     275
FMAILER  O   F      50          LPRINTER
FMAILER   18FL 180L
IRPGVEN     01
I                                        4  53 NAME
I                                      104 143 STREET
I                                      144 165 CITY
I                                      174 175 STATE
I                                      176 1800ZIP
OMAILER  D 1  3   01
O                           NAME       50
O            D 1      01
O                           STREET     40
O            D 1      01
O                           CITY       22
O                           STATE      24
O                           ZIP        31
```

Figure 3-14 ▪ A Typical RPG II Program Used to Generate Mailing Labels


VAX Rdb/VMS (ELN)
VAX Language Sensitive Editor (LSE)
The VMS Operating System
VAX DECShell
VAX DECCode Management System (CMS)

Chapter 4 ▪ VMS Services

▪ Chapter Overview

The VMS operating system provides you with a number of important services, each of which has a unique role in streamlining the program development effort. VMS services are: the Digital Command Language (DCL), VAX Record Management System (RMS), the VAX Runtime Library (RTL), and VMS System Services.

The command language (DCL) provides a consistent interface with which you can access the VMS operating system. Many options exist to provide detailed levels of control where needed. VAX RMS, the RTL, and VMS System Service facilities help reduce application design time by taking many of the specific data management, peripheral interfacing, and networking elements out of the application program. These four VMS Services are covered in this chapter.

Topics include:

- The Digital Command Language (DCL)
- VAX Record Management Services (RMS)
- The VMS Runtime Library (RTL)
- The VMS System Services

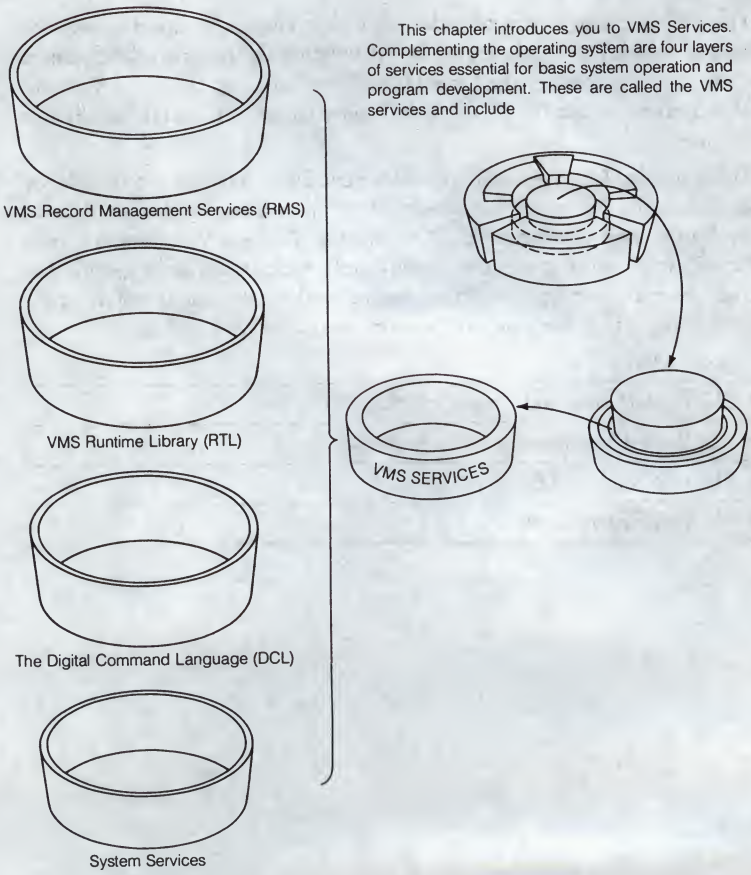


Figure 4-1 ▪ Overview of Chapter 4

▪ VMS Record Management Services (RMS)

The VAX RMS facility is the standard Digital software for data management services and provides an interface at the application-program level to record/file management functions.

RMS is a powerful collection of routines that provide application programmers with a device-independent method for the extensive storage, retrieval, and modification of data. Complex file manipulation is easily achieved through RMS facilities. You may select from several file organizations and file access techniques — each of which is suited to a particular application. These can range from the simplest sequential search of a sequentially organized file to a sophisticated keyed access of an indexed file based on several alternate key fields. RMS supports sequential, relative, and multikey indexed-sequential file organizations.

For information about the use of RMS services in a VAXcluster system, see Chapter 5, VAXcluster Software, of the *VMS System Software Handbook*.

Files

A file is a collection of related information whose requirements are established by the nature of application programs needing the information. A company might maintain information regarding an employee in one file, and product information in another.

Each record in a file is subdivided into discrete pieces of information known as data fields. You define the number, location within the record, and logical interpretation of each. It is necessary, therefore, to embed the relationship of records and fields into an application program. When a program is written using RMS, you don not have to be aware of such logical relationships. Programs either build records and pass them to RMS for storage in a file or issue requests for records while RMS performs the necessary operations to retrieve the records from a file.

The purpose of RMS, then, is to ensure that every record written into a file can subsequently be retrieved and passed to a requesting program as a single logical unit of data. The structure, or organization, of a file establishes the manner in which RMS stores and retrieves records. The way a program requests the storage or retrieval of records is known as the access mode. Legal access modes depend on the file organization.

RMS Provides Three Record-Access Modes

The various methods of retrieving and storing records in a file are called access modes. A different access mode can be used to process records within the file each time it is opened. A program can also change access mode during the processing of a file, by a procedure known as a dynamic access.

RMS provides three record-access modes:

- Sequential
- Random
- Record's file address (RFA)

- SEQUENTIAL ACCESS MODE
  Sequential access mode can be used with any RMS file. Sequential access means that records are retrieved or written in a particular sequence. The organization of the file establishes this sequence.

- RANDOM ACCESS MODE
  In random access mode, the program, rather than the organization of the file, establishes the order in which records are processed. Each program request for access to a record operates independently of the previous record accessed. Associated with each request in random mode is an identification of the particular record of interest. Successive requests in random mode can identify and access records anywhere in the file.

- RECORD'S FILE ADDRESS (RFA) ACCESS MODE
  Record's file address (RFA) access mode can be used with any file organization as long as the file resides on a disk device. This access mode is further limited to retrieval operations only. Like random access mode, however, RFA access allows a specific record to be identified for retrieval.

  As the name suggests, every record within a file has a unique address. The actual format of this address depends on the organization of the file. In all instances, however, only RMS can interpret this format.

  The most important feature of RFA access is that the address (RFA) of any record remains constant while the record exists in the file. After every successful read or write operation, RMS returns the RFA of the subject record to the program. The program can then save this RFA to use again to retrieve the same record. RFAs can be saved and used at any later time.

- DYNAMIC ACCESS
  Dynamic access is not strictly an access mode. Rather, it is the capability to switch from one access mode to another while processing a file. The only limitation is that the file organization must support the access mode selected.

As an example, dynamic access can be used effectively immediately following a random or RFA access mode operation. When a program accesses a record in one of these modes, RMS establishes a new current position in the file. Programs can then switch to sequential access mode. By using the randomly accessed record (rather than the beginning of the file) as the starting point, programs can retrieve succeeding records in the sequence established by the file's organization.

RMS File Attributes

The most important attribute of any RMS file is its organization. A file for use in a particular application can be tailored by making the proper selection of this and other attributes. RMS file organization is sequential, relative, or indexed. In addition to file organization, you can choose from among the following attributes:

- Storage medium where the file resides
- File name and protection specification of the file
- Format of records
- Size of a particular storage structure, known as the bucket, within relative and indexed files
- Definition of keys for indexed files

- STORAGE MEDIA

  Selection of a storage medium on which RMS builds a file is related to the organization for the file. Permanent sequential files can be created on disk devices or ANSI magnetic tape volumes. Transient files can be written on devices such as lineprinters and terminals. Unlike sequential files, relative and indexed files can reside only on disk devices.

- FILE SPECIFICATION

  The name assigned to a new file enables RMS to find the file on the storage medium. RMS allows a protection specification to be assigned to a file at the time it is created.

  When a file is created, you must provide the format and maximum size specification for the records the file will contain. The specified format establishes how each record appears physically in the file on a storage medium. The size specification allows RMS to verify that records written into the file do not exceed the length specified when the file was created.

- RMS RECORD FORMATS

  RMS formats include:

  - Fixed
  - Variable-with-fixed-control (VFC)
  - Stream

Program Operations on RMS Files

After a file has been created, a program can access the file and store and retrieve data.

When a program accesses the file as a logical structure (for example, a sequential, relative, or indexed file), it uses record I/O operations such as add, update, read and delete record. The organization of the file determines the types of record operations permitted.

If the record accessing capabilities of RMS are not used, programs can access the file as an array of virtual blocks. To process a file at this level, programs use a type of access known as block I/O.

- FILE PROCESSING

  At the file level, before beginning record processing, a program can:

  - Create a file
  - Open an existing file
  - Modify file attributes
  - Extend a file
  - Close a file
  - Delete a file

  Once a program has opened a file for the first time, it has access to the unique internal ID for the file. If the program intends to open the file subsequently, it can use that internal ID to open the file and avoid any directory search.

- FILE ORGANIZATION AND SHARING

  With the exception of magnetic tape files, which cannot be shared, every RMS file can be shared by any number of programs that are reading, but not writing, the file. Sequential files on disk can be accessed by a single writer or shared by multiple readers. Relative and indexed files, however, can be shared by multiple readers and multiple writers. A program can read or write records in a relative or indexed file while other programs are similarly reading or writing records in the file. Thus, the information in such files can be changing while programs are accessing them.

- PROGRAM SHARING

  A file's organization establishes whether it can be shared for reading with a single writer or for multiple readers and writers. A program specifies whether such sharing actually occurs at runtime. You control the sharing of a file through information the program provides RMS when it opens the file. First, a program must declare what operations (that is, read, write, delete, update) it intends to perform on the file. Second, a program must specify whether other programs can read the file or both read and write the file concurrently with the first program.

  The combination of these two types of information allows RMS to determine if more than one program can access a file at the same time. Whenever a program's sharing information is compatible with the corresponding information another program provides, both programs can access the file concurrently.

- RECORD LOCKING

  RMS can lock records to control operations to a relative or indexed file that more than one record stream within a process, or more than one process, can access simultaneously. The purpose of this facility is to ensure that a program can add, delete, or modify a record in a file without another program simultaneously accessing the same record.

  When a program opens an indexed or relative file with the declared intention of writing or updating records, RMS locks any record accessed by the program. This locking prevents another program from accessing that record until the program releases it. The lock remains in effect until the program accesses another record. RMS then unlocks the first record and locks the second. The first record is then available for access by another concurrently executing program.

  A program can also select a manual unlocking mode, in which all records accessed by the program remain locked until they are explicitly unlocked by calls to RMS. Additionally, a program may request that no record locking be done.

- RECORD I/O PROCESSING

The organization of a file, defined when the file is created, determines the types of operations that the program can perform on records. Depending on file organization, Record Management Services permits a program to perform the following record operations:

- Get a record — RMS returns an existing record within the file to the program
- Put a record — RMS adds a new record that the program constructs to the file. The new record cannot replace an already existing record
- Find a record — RMS locates an existing record in the file. It does not return the record to the program, but establishes a new current position in the file
- Delete a record — RMS removes an existing record from the file. The delete record operation is not valid for sequential file organizations
- Update a record — The program modifies the contents of a record read from the file. RMS writes the modified record into the file, replacing the old record. The update record operation is generally not valid for sequential file organizations, except for precise replacement of records in those with fixed-length records.

RMS Utilities

The RMS procedures are complimented by a File Definition Language (FDL) and a number of utilities designed especially for RMS file creation and maintenance. They are called directly through DCL, and include:

- CONVERT
- CONVERT/RECLAIM
- EDIT/FDL
- CREATE/FDL
- ANALYZE/RMS_FILE

The File Definition Language is a special purpose language used to describe file organizations for data files. These specifications are then used by the RMS utilities and library routines to create data files and other data structures.

Using RMS

RMS is a powerful tool for handling input/output tasks. Whether you simply need to have a program read input lines from a terminal, or need full write-sharing capability with record locking — allowing multiple processes to access and update records in the same files simultaneously — RMS can simplify and handle the task. Of course, more complex operations may require a number of parameters and allow specification of many more; nevertheless, all of the basic RMS services use one of two control structures as input for their operation. The File Access Block (FAB) contains only fields relevant to file operations, such as the creation of a new file or opening an existing one. The Record Access Block (RAB) contains parameters necessary to perform record operations, such as record retrieval and update, on records within a file. The following table illustrates this division.

Table 4-1 ▪ Comparison of RAB and FAB Parameters For Record Operations

Category	Macro Name	Service
File Processing (FAB = address)	$CREATE	Creates and opens a new file
	$OPEN	Opens an existing file and initiates file processing
	$CLOSE	Terminates file processing and closes the file
Record Processing (RAB = address)	$CONNECT	Associates and connects a RAB to the file
	$GET	Retrieves a record from a file
	$PUT	Writes a new record to a file
	$UPDATE	Rewrites an existing record in a file

The brief program listing that follows, with comments, demonstrates the ease and simplicity of using RMS to achieve an I/O operation. Several different runs of the program follow. It reads a sequential file containing ASCII text and uses a Runtime Library routine to print the text on your terminal.

```
1 Buffer  .blkb          100              ;allocate a 100 byte buffer
2 Buff_desc:                              ;descriptor for buffer
3         .long          0                ;length will be set at runtime
4         .long          Buffer           ;address of buffer
5
6 My_fab: $FAB           FNM=<INFILE>     ;File access block
7
8 My_rab: $RAB           FAB=My_fab,-     ;Record access block
9                        UBF=Buffer,-
10                       USZ=100
11
12 Start: .word          0
13
14        $OPEN          FAB=My_fab       ;open the file
15        BLBC           R0,Close file    ;exit on error
16
17        $CONNECT       RAB=My_rab       ;connect for record operations
18        BLBC           R0,Exit          ;exit on error
19
20 Get_record:
21        $GET           RAB=My_rab       ;get the first record
22        BLBC           R0,Exit          ;exit on error
23
24        MOVW           My_rab + rab$w_rsz,Buff_desc
                                          ;store length of record in
                                           desc
25        PUSHAB         Buff_desc        ;push descriptor address for
                                           output
26        CALLS          #1,G^LIB$PUT_OUTPUT ;print the record
27        BRB            Get_record       ;go back and get the next
                                          record
28 EXIT:
29        $CLOSE         FAB=My_fab       ;and close the file
30
31         RET
32
33        .end            Start
```

Figure 4-2 ▪ A Sample Program Using RMS

▪ The Digital Command Language (DCL)

A single command language, the Digital Command Language (DCL), provides you, as a user of a VAX/VMS system, with an extensive set of commands for:

- Interactive program development.
- Device and data file manipulation.
- Interactive and batch program execution and control.
- Operational control.

Commands exist for program development and execution, for resource allocation, environmental control, job control, file maintenance, utilities, and operational control.

- Program development and execution commands include commands to invoke each compiler, the assembler, the editor, and the linker, as well as to run any prelinked program.
- Resource allocation commands include the ability to allocate and deallocate devices and mount and dismount volumes.
- Environmental commands include assign and deassign logical names and set and show parameters such as job status, terminal type, and default directory.
- Job control commands include the ability to continue and stop execution, a GOTO command to transfer control, and IF and ON commands to specify error handling.

DCL also includes commands to login and logout, to submit batch jobs, to send messages to the operator, and to prompt you for input. File maintenance commands include append to files, copy, create, and delete files, list directories, initialize volumes, print and type files, and rename files.

Topics covered in this section include:

- Command procedures
- Commands
- Terminal function keys

Command Format

DCL commands are composed of English words. Any file name can be given a logical name for mnemonic reference. Command parameters can be supplied on the same line as the command verb. Missing parameters will be prompted for by the VMS command interpreter. To make it easier to learn the VMS system, an extensive HELP facility gives guidance on the use of commands and the meaning of system messages.

Typical VMS commands are brief because of the extensive use of defaults. You also can define additional commands and use them just as the system-defined commands are used. All command verbs and qualifiers can be abbreviated to the shortest unique form.

File specifications can be as simple as the user-given name of the file only, or as complex as a full specification of network node, device (including type, controller, and unit), directory, file name, file type, and version number. Logical names can be defined for complex file specifications so that repetitive typing can be avoided.

Command Procedures

A command procedure is a file that contains a list of DCL commands. When you execute a command procedure, the DCL command interpreter reads the file and executes the commands in it.

Command procedures can be used to automate a sequence of commands that you use frequently. For example, if you always issue a DIRECTORY command after you move to a subdirectory, you can write a simple command procedure to issue the SET DEFAULT and DIRECTORY commands for you as illustrated in the following example.

```
$ SET DEFAULT [SMITH.ACCOUNTS]
$ DIRECTORY
```

Instead of issuing each command, you could greatly simplify the operation and reduce the number of keystrokes by using a command procedure (named GO_DIR.COM) that would be executed when you type:

```
$ @GO_DIR
```

This command tells the DCL command interpreter to read the file GO_DIR.COM and to execute the commands in the file. Therefore, the command interpreter sets your default directory to [SMITH.ACCOUNTS] and issues the directory command.

You can write complex command procedures that resemble programs written in high-level programming languages. In this sense, a command procedure provides a method of writing programs in the Digital Command Language.

You can, for example, revise GO_DIR.COM to allow you to move to any directory and obtain a list of the files in the directory and obtain a list of the files in the directory:

```
$ INQUIRE DIR_NAME "Directory name:
$ SET DEFAULT 'DIR_NAME'
$ DIRECTORY 'DIR_NAME'
```

When you execute GO_DIR.COM, the INQUIRE command prompts for a directory name, the SET DEFAULT command moves you to the directory, and the DIRECTORY command lists the file names.

- FORMATTING COMMAND PROCEDURES

  Use a text editor (or the DCL command CREATE) to create and format a command procedure. When you name the command procedure, use the default file type COM. If you use this default file type, you do not have to include a file type when you execute the procedure with the "@" command.

  Command procedures contain DCL commands that you want the DCL command interpreter to execute and data lines that are used by these commands. Commands must begin with a dollar sign ($). You can start the command string immediately after the dollar sign, or you can place one or more spaces or tabs before the command sting to make it easier to read.

  Data lines, unlike commands, do not begin with a dollar sign. Data lines are used as input data for commands (or images.) Data lines are used by the most recently issued command; these lines are not processed by the DCL interpreter.

  The following example illustrates command lines in a command procedure.

```
$ MAIL
SEND
THOMAS
MY MEMO
Do you have a few minutes to talk about the ideas
I presented in my memo?
$
$ SHOW USERS THOMAS
```

  In the preceding example, the first line is a command and must start with a dollar sign. The next lines are data lines that are used by the MAIL utility; these lines must not start with a dollar sign. Note that data lines must correspond to the way the image (invoked by the command) expects the data. Therefore, the data lines provide a MAIL command (SEND), a recipient (THOMAS), a subject (MY MEMO) and the text of of the mail message. When the command interpreter finds a new line that begins with a dollar sign, the MAIL utility is terminated.

  For more information on the Digital Command Language, see Chapter 2 of the *VMS System Software Handbook.*

▪ The VMS Runtime Library

The VMS Runtime Library is a set of language-independent procedures that establish a common runtime environment for application programs written in any VAX language in the Common Language Environment. Because the RTL adheres to the VAX calling standard, procedures can be called from programs written in VAX languages that also follow the calling standard. Therefore, application programs can be composed of modules written in many different languages, including the VAX assembly language.

Features of the Runtime Library

The VMS RTL provides the following features and capabilities.

- RTL procedures perform a wide range of general-utility operations; you can call an RTL procedure instead of writing code to perform the same operation.
- The results of a particular procedure are the same, no matter what language calls it.
- RTL procedures use VMS Record Management Services for file I/O.
- Library procedures can be updated without revising programs that call its shared modules.

Organization of the Runtime Library

The VMS Runtime Library contains several facilities. These facilities are groups of procedures that perform related operations. The VMS library facilities are listed in Table 4-2.

Table 4-2 ▪ VMS Runtime Library Facilities

Facility	Description
LIB$	General purpose procedures
MTH$	Mathematics procedures
SMG$	Screen management procedures
STR$	String manipulation procedures
OTS$	Language-independent support procedures
BAS$	BASIC-specific support procedures
COB$	COBOL-specific support procedures
FOR$	FORTRAN-specific support procedures
PAS$	Pascal-specific support procedures
PLI$	PL/I-specific support procedures
RPG$	RPG-specific support procedures

These facilities are divided into general-purpose and language-support facilities. General-purpose procedures are intended to be called explicitly to perform common procedures, while language-support procedures are intended to be called implicitly by language compilers and compiled code. Language-support procedures are further divided into language-specific and language-independent support procedures.

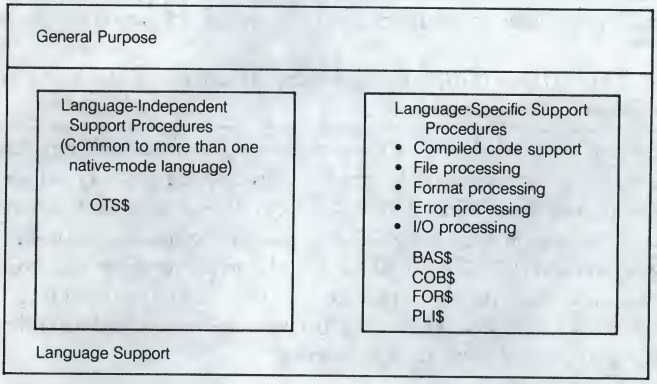


Figure 4-3 ▪ General-Purpose and Language-Support RTL Procedures

Functional Listing of VMS RTL Procedures

These facilities can be categorized by function as listed in Table 4-3.

Table 4-3 ▪ Functional Grouping of VMS RTL Facilities

Function	RTL Facility
General Utility Procedures	LIB$ STR$ OTS$
Mathematical Procedures	MTH$ OTS$
Process-wide Resource Allocation Procedures	LIB$ STR$ OTS$
Signaling and Condition Handling Procedures	LIB$
Special Application Procedures	LIB$

GENERAL PURPOSE PROCEDURES - In most cases, application programs call these procedures using explicit CALL statements or function references.

GENERAL UTILITY (LIB$) - General utility procedures provide a wide range of functions for your convenience.

- Common I/O procedures — These procedures perform such functions as getting records from the current input device (LIB$GET_INPUT) and sending them to the output device (LIB$PUT_OUTPUT), executing DCL commands from a running program (LIB$SPAWN), getting the command line from a foreign command (LIB$GET_FOREIGN), and copying strings to and from the process's common storage area (LIBPUT_COMMON, LIBGET_COMMON). I/O control procedures are also available to customize printer output and translate logical names.

- Terminal independent screen procedures — These procedures provide a high-level language interface to the video terminal. They send text to the screen, move the cursor to the desired position, erase text from the screen, and manipulate the screen buffer.

- Data type conversion procedures — These procedures perform conversions between one VAX data type and another (for example, text to D-floating and decimal to binary).

- Variable bit field instruction procedures — LIB$INSV inserts and extend variable bit fields. LIB$FFS searches a bit field for the first set bit.

- Performance measurement procedures — These procedures provide a facility for timing, counting I/O operations, and counting page faults.

- Date/time utility procedures — These procedures return the system date or time in several forms.

- CRC procedures — LIB$CRC and LIB$CRC_TABLE permit you to calculate the cyclic or redundancy check for a data stream

- Multiple precision arithmetic procedures — LIB$ADDX and LIB$SUBX add and subtract signed two's-complement integers of arbitrary length.

Runtime Library procedures also permit the high-level language programs to use the following VAX hardware instructions:

- Extended multiply and modulus arithmetic — EMUL, EDIV, EMOD

- Evaluate polynomials — POLYF, POLYD, POLYG, POLYH

- Insert and remove queue entry — INSQHI, INSQTI, REMQHI, REMQTI

MATHEMATICAL FUNCTIONS (MTH$) - The mathematical library consists of over 200 standard procedures to perform common mathematical functions. These functions include:

- Floating-point mathematical functions: trigonometric, logarithmic, and square root
- Complex functions: absolute values, conjugation, trigonometric, arithmetic, exponentiation, return imaginary part of complex number, return real part of complex number, make complex from floating-point, logarithmic, and square root.
- Exponentiation on floating-point, word, longword, and complex data
- Random number generator
- Processor-defined mathematical procedures: includes both the intrinsic and basic external functions defined in ANSI FORTRAN.

  They include routines that perform conversions between floating-point and integer data and a large number of miscellaneous data types

RESOURCE ALLOCATION PROCEDURES (LIB$, STR$, OTS$) - The Resource Allocation Section includes all procedures that allow allocation of process-wide resources. Such resources include the following:

- Virtual memory — one procedure to allocate and one to deallocate arbitrarily sized blocks of process virtual memory
- Logical unit numbers — allow logical numbers to be allocated in a modular manner
- Event flags — same as logical unit numbers

In most cases, the resource allocation procedures should used to allocate process-wide resources in order for all library, DIGITAL, and customer-written procedures to work together properly within a process.

SIGNALING AND CONDITION HANDLING - The VMS condition handling facility is a collection of library procedures and system services that provides a unified and standardized mechanism for handling errors internally in the operating system, the Runtime Library, and application programs. In some cases, the mechanism is also used to communicate errors across these interfaces. In particular, all error messages are printed using this mechanism. When an error condition is signaled, the process stack is scanned in reverse order. Establishing a handler provides you with control over fix-up, reporting, and flow of control on errors. It provides the system and library messages in order to give a more suitable application-oriented user interface.

LANGUAGE-INDEPENDENT SUPPORT (OTS$) - The language support libraries support the code generated inline by compilers. As such, most of the procedures are called implicitly as a consequence of a language construct you specified, rather than being called explicitly with a CALL statement. Those language support procedures that are independent of higher-level languages use the facility prefix OTS$. They include:

- Language-independent initialization and termination
- Error and exception-condition processing procedures
- Data type conversion

LANGUAGE-SPECIFIC SUPPORT - Each of the language-specific support libraries is generally composed of:

- I/O processing procedures
- File processing procedures
- Compiled-code support procedures
- Compatibility procedures
- System procedures

STRING PROCESSING (STR$) - The string processing procedures allocate and deallocate dynamic strings. They also perform a wide variety of string manipulation functions, such as comparing, locating a character, concatenating, extracting a substring, performing arithmetic operations on decimal strings, and translating ASCII to EBCDIC code.

SCREEN MANAGEMENT PROCEDURES (SMG$) - The Screen Management facility allows application programs to be coded without regard to physical devices performing I/O. Instead of writing directly to a physical screen, your program writes to a virtual display. Similarly, user programs perform input from a virtual keyboard instead of a physical keyboard.

The screen management facility provides two important services:terminal independence and composition aids.

TERMINAL INDEPENDENCE - The screen management procedures provide terminal independence by allowing commonly needed screen functions to be performed without concern for the type of terminal being used. All operations are performed by calling a procedure that converts the caller's terminal-independent request (for example, scroll a part of the screen) into the appropriate sequence of code needed to perform that action. If the terminal being used does not support the requested operation in the hardware, the screen management procedures emulate the action in software.

COMPOSITION AIDS - The screen management procedures simplify the composition of complex images on a terminal screen. For example, if an application program was directed to solicit your input from one part of the screen, display results to another, and maintain a status display in another, procedures from the Screen Management Facility enable code for each of these operations to be written independently of the other.

SYSTEM PROCEDURES - VAX programs written in the higher-level languages can call the operating system directly. However, since some languages cannot easily pass arguments in the form that system services require, and some languages use data types that system services cannot properly handle (that is, dynamic strings), some LIB$ routines have been provided to handle the input and output arguments correctly.

▪ VMS System Services

VMS System Services are procedures provided by the VMS operating system. These procedures

- Control resources available to processes.
- Provide for communication among processes.
- Perform basic operating system functions (I/O coordination, for example.)

VMS system services are available for general use by logged-in users or can be called by an application program. For example, the operating system creates your process with the Create Process system service ($CREPRC) when you log in. $CREPRC can also be called from an application program to create a subprocess to perform certain functions for that program.

System services are divided into 11 functional groups. Table 4-4 lists each group of services and the general function of that group.

Table 4-4 ▪ The Functional Grouping of VMS System Services

Service Group	Function
Security	Security system services provide various mechanisms to enhance and control system security.
Event-Flag	Event-flag services clear, set, and read event flags and can place a process in a wait state pending the setting of those flags.
AST	Process execution can be interrupted by events for the execution of designated subroutines. These software interrupts are called Asynchronous System Traps (ASTs). AST system services are provided so that a process can control the handling of these interrupts.
Logical Names	Logical name services provide a generalized technique for maintaining and accessing character string logical name and equivalence name pairs.
Input/Output	I/O system services bypass VAX Record Management Services (RMS) and perform input and output operations directly at the device driver level. I/O system services include: * Perform logical, physical, and virtual input/output operations. * Format output lines converting binary-numeric values to ASCII strings and substitute variable data in ASCII strings. * Perform network operations. * Queue messages to system processes.
Process-Control	Process-control system services allow you to create, delete, and control the execution of processes.
Timer and-Time-Conversion	Timer services schedule program events for a particular time of the day, or after a specified interval of time has elapsed. The time-conversion services allow you obtain and format binary time values for use with the timer services.

(continued on next page)

Table 4-4 ▪ The Functional Grouping of VMS System Services (Cont.)

Service Group	Function
Condition-Handling	Condition handlers are procedures that can be designated to receive control when a hardware or software exception/condition occur. Condition-handling services designate condition handlers for special purposes.
Memory-Management	Memory-management services give you control over an application program's virtual address space. These services: * Allow an image to increase or decrease the amount of virtual memory available. * Control the paging and swapping of virtual memory. * Create and access files that contain sharable code and data. Change-Mode Change-mode services changes the access mode of a process. These services are used primarily by the operating system.
Lock-Management	Lock-management services make it possible for co-operating processes to synchronize their access to shared resources.

Calling System Services

At runtime, an application program calls a system service and passes control of the process to it. Upon execution of the system service, the services returns control to the program and also returns a condition value. The program then analyzes the condition value, and determines the success of failure of the system service call, and alters program execution flow as required.

- VMS SYSTEM SERVICES AND SYSTEM INTEGRITY

Many system services are available and suitable for application programs, but the use of some services must be restricted to protect the performance of the system and the integrity of your process.

The creation of permanent mailboxes, for example, requires the use of system-dynamic memory. Therefore, the unrestricted use of permanent mailboxes could decrease the amount of memory available to other system users. Thus, the use of this system service is controlled by the assignment of a specific user privilege.

The system manager grants the use of privileges to use these protected system services and controls that privilege through the use of User Authorization File (UAF). This file contains a specific list of user privileges and resource quotas.

When you log in, the privileges and quotas you have been assigned are associated with the process created on your behalf. These privileges and quotas are applied to every image that the process executes.

A privilege list is checked when an image in your process issues a call to a protected system service. If you have been granted the specific privilege required, the image is allowed to execute the system service. If not, a condition value indicating the privilege violation is returned.

VMS Security System Services

The VMS Security System Services provide the system manager or the application programmer with many security-based resources of the VMS operating system. These resources allow you to protect and and fine tune the security of your computing environment.

These services include facilities to

- Create and maintain a rights database.
- Create and translate access-control entries.
- Modify a process-rights list.
- Check access protection.
- Provide a security-erase pattern for disks.
- Control access to magnetic tapes.

VMS Event-flag System Services

Event flags are maintained by the VMS operating system for general programming use. Programs can use event flags to perform a variety of signaling functions. The VMS event-flag system services clear, set and read event flags. They can also place a process in a wait state pending the setting of an event flag or flags.

Event flags can also be used by more than one process as long as the cooperating processes are in the same group. Thus, if you have developed an application that requires the concurrent execution of several processes, you can use event flags to establish communication among them and to synchronize their activity.

- EVENT-FLAG NUMBERS AND EVENT-FLAG CLUSTERS

  Each event flag has a unique decimal number; event-flag arguments in system-service calls refer to these numbers. For example, if you specify event flag 1 in a call to the $QIO system service, then event flag number 1 is set when the I/O completes.

  Groups of event flags are manipulated by organizing them into event-flag "clusters." Each "cluster" is made up of 32 flags.

  There are two types of "clusters," local event-flag clusters and common event-flag "clusters."

  - Local event-flag "clusters" can only be used internally by a single process. Local "clusters" are automatically available to each process.
  - A common event-flag cluster can be shared cooperating processes in the same group. Before a process can refer to a common event-flag cluster, it must explicitly associate with the cluster.

Some system services set event flags to indicate the completion or the occurrence of an event; the calling program can test the flag.

AST (Asynchronous System Trap) Services

Some system services allow a process to request that it be interrupted when a particular event occurs. Sine the interrupt occurs asynchronously (out of sequence) with respect to the process's execution, the interrupt mechanism is called an asynchronous system trap (AST). The trap provides a transfer of control to a user-specified procedure that handles the event.

The system services that use the AST mechanism accept the address of an AST service routine as an argument. That is, the routine to be given control when the trap occurs.

The following list of services use ASTs.

- Declare AST ($DCLAST)
- Enqueue Lock Request ($ENQ)
- Get Device/Volume Information ($GETDVI)
- Get Job/Process Information ($GETJPI)
- Get System-Wide Information ($GETSYI)
- Queue I/O Request ($QIO)
- Set Timer ($SETIMR)
- Set Power Recovery AST ($SETPRA)
- Update Section File on Disk ($UPDSEC)

For example, if you call the Set Timer ($SETIMR) service, you can specify the address of a routine to be executed when a time interval expires or at a particular time of day. The service schedules the execution of the routine and returns; the program image continues executing. When the requested time event occurs, the system "delivers" an AST by interrupting the process and calling the specified routine.

Logical Name Services

The VMS logical name services provide a technique for manipulating and substituting character string names. Logical names are commonly used to specify devices or files for input or output operations. You can use logical names to communicate information between processes by creating a logical name in a logical name table that is accessable by another process.

The VMS operating system services establish logical names for general application purposes. The system also performs special logical name translation procedures for names associated with I/O services and with services that can deal with facilities located in shared (multiport) memory.

Input/Output System Services

When writing application programs, there are two basic input/output services available to the application developer, VAX Record Management Services (RMS) and VMS I/O system services. VAX RMS provides a set of routines for general purpose, device-independent functions such as data storage, retrieval, and modification.

The I/O system services permit you to use the I/O resources of the operating system directly, in a device-dependent manner. I/O services also provide some specialized functions not available in VAX RMS. Using I/O services requires more knowledge on your part than if you used VAX RMS. However, using these services result in more efficient input/output operations.

Process-Control Services

When you log into the system, the system creates a process for the execution of program images. You can create another process to execute an image by issuing the RUN or SPAWN command using any of the qualifiers that pertain to process creation. You can also write a program that creates another process to execute a particular image.

Process control services allow you to create processes and to control a process or group of processes. They allow you to

- Create subprocesses and detached processes.
- Control the execution of a process.
- Facilitate control and communication between processes.
- Control the hibernation or suspension of a process.
- Control image-exit and exit handlers.
- Control process deletion.

VMS Timer and Time-Conversion System Services

Many applications require the scheduling of program activities based on clock time. Under VAX/VMS, an image can schedule events for a specific time of day or after a specified time interval.

You can use VMS timer services to schedule, convert, or cancel events. For example, you may use the time services to

- Schedule the setting of an event flag or the queuing of an AST for the current process, or cancel a pending request that has not yet been honored.
- Schedule a wake-up request for a hibernating process, or cancel a pending wake-up request that has not yet been honored.

The timer services require you to specify the time in a 64-bit format. To work with the time in different formats, you can use time conversion services to

- Obtain the current date and time in an ASCII string or in system format.
- Convert an ASCII string into the system-time format.
- Convert a system-time value into an ASCII string.
- Convert the time from system format into integer values.

VMS Condition-Handling System Services

A condition handler is a procedure given control by the operating system when an exception occurs. An exception is an event, detected by the hardware or software, and that interrupts the execution of an image. Examples of exceptions include arithmetic overflow and reserved opcode or operand faults.

If you determine that a program needs to be informed of particular exceptions, you can write and specify a condition handler. This condition handler, which receives control when any exception occurs, can test for specific exceptions.

VMS Memory-Management System Services

The VAX/VMS memory-management routines map and control the relationship between physical memory and a process's virtual address space. These activities are, for the most part, transparent to you as a user and to your programs. However, a program can be run more efficiently by allowing it to explicitly control its use of virtual memory.

VMS Lock-Management System Services

The VMS lock-management system services allow cooperating processes to synchronize their access to shared resources. This is accomplished by providing a common data area in which processes can lock a specified resource by name. All processes that access the resources must use the VMS lock-management services to be effective.

The lock-management services provide a queuing mechanism that allows processes to with in a queue until a particular resource is available.

Chapter 5 ▪ VMS Program Development Utilities

▪ Chapter Overview

Besides the services discussed in Chapter 4, VMS also provides you with many powerful program development utilities, such as text editors, debugging facilities, program module linkers and many other sophisticated tools that can be used in every facet of the program development life cycle. This chapter gives you a overview of many of our more important program development utilities.

VMS text editors gives you the ability to create high-level source code programs quickly and efficiently. The DSR text-formatting utility extends basic editor capabilities into a highly sophisticated text processing system.

Program debugging, high-speed sorting and merging of data, and the linking of object modules into executable images are a few of the major functions these utilities perform.

Products covered in this chapter include

- The VAX/VMS Symbolic Debugger Utility.
- The VMS SORT/MERGE Utility.
- The VMS LINKER Utility.
- The VMS LIBRARIAN Utility.
- Other VMS program development utilities.
- VAXTPU (Text processing utility)
- The EDT text editor
- The DSR Text Formatting Utility

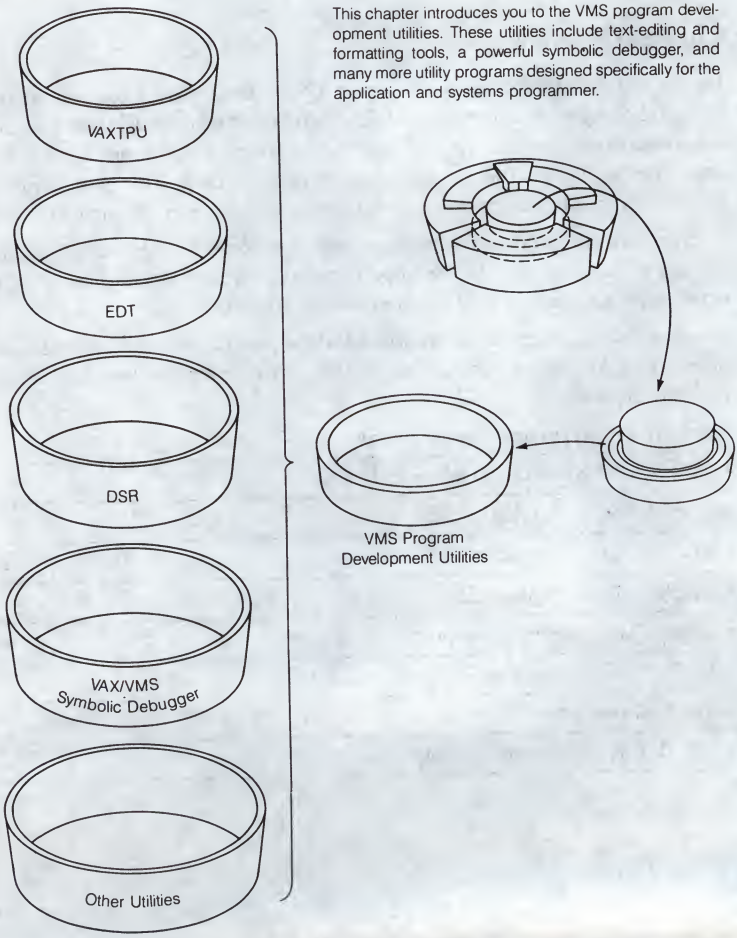


Figure 5-1 ▪ Overview Of Chapter 5

▪ The VAX/VMS Symbolic Debugger

The VAX/VMS Symbolic Debugger is a powerful and flexible tool used to find errors in source code programs.

The VAX/VMS Symbolic Debugger

- Is interactive. You can execute Debugger commands from your terminal and see their effects immediately.
- Is symbolic. You can refer to program locations by the symbols you used for them in your program.
- Supports many languages. You use the Debugger in the language of your source program. If your application is written in more than one language, you can change from one language to another within the course of a debugging session.
- Permits a variety of data forms and types for entry and display.
- Allows you to select and display your program's language statements.
- Has a screen mode that provides multiple windows for screen-oriented debugging.
- Has a debugger-defined keypad key definitions for your terminal's numeric keypad.
- Gives online help.
- Supports the VAX Language-Sensitive Editor

The VAX/VMS Symbolic Debugger Is Interactive

When using the VAX/VMS Symbolic Debugger, you run the program to be debugged interactively, at your terminal under Debugger control. When the program starts to execute, VAX/VMS Symbolic Debugger gains control before the program and prompts for VAX/VMS Symbolic Debugger commands. These commands are entered to control the execution of your program by interrupting that execution at specific intervals. This momentary pause allows you to enter more VAX/VMS-Symbolic-Debugger commands and to gather additional information about the current state of your program.

The VAX/VMS Symbolic Debugger Utility Is Symbolic

With the VAX/VMS Symbolic Debugger, you can reference program locations to by symbolic names. For example, if you wish to interrupt the execution of your program at routine BILL, the SET BREAK BILL command will look up the address for the symbol BILL and stop when the PC hits that address.

Likewise, EXAMINE X will look up the address of the symbol X and display the contents of that address. You don't need to know where X or BILL are in memory, the Debugger knows for you.

The VAX/VMS Symbolic Debugger Is Multilingual

The Debugger is a multilingual debugger, supporting many VAX languages. These include:

VAX Ada	VAX BLISS
VAX BASIC	VAX C
VAX COBOL	VAX FORTRAN
VAX MACRO	VAX PASCAL
VAX PL/I	VAX RPG II

The VAX/VMS Symbolic Debugger is strictly an object program debugger, debugging programs at run time, after they have been compiled and linked. It is used to debug only compiled languages — not interpreted ones.

Being multilingual means that the Debugger understands the following characteristics of each supported language.

- How symbol names are composed in the language. It knows how identifiers are formed and how compound names are constructed. For example, it accepts A(2)→B as valid PL/I syntax and A[2].B as valid PASCAL syntax.
- How language expressions are interpreted. It knows what operators are allowed and what their syntax and semantics are.
- How and when type conversions are done in the language. This is part of understanding how to interpret expressions and is needed to do assignments properly.
- How values are displayed in the language. For example, how enumeration type values are displayed in PASCAL and how numeric values are displayed in COBOL.
- How the language scope rules work. It knows how to look up a symbol name in a specified scope according to language rules.

- VMS DEBUGGER FEATURES AND COMMANDS

The VAX/VMS Symbolic Debugger command language defines the capabilities of this debugger and constitutes the user interface. The following is a brief synopsis outlining the main features and commands of VAX/VMS Symbolic Debugger.

- The EXAMINE command retrieves and displays the contents of a specified program location. The location is typically a variable and the display is formatted according to the variable's type.
- The DEPOSIT command stores a new value into a specified location, again usually a variable.
- The EVALUATE command permits expressions in the language to be evaluated and the results displayed.
- The SET WATCH command causes the program to stop when a specified data location is altered.
- The STEP command stops the program when the next instruction or the next source line is reached.

- USER-INTERFACE FEATURES OF THE VMS DEBUGGER UTILITY

These features are

- Screen Mode – The debugger uses the full terminal screen for output. The screen can be divided into multiple windows which can be scrolled up and down as well as right and left. These windows may contain program source code, debugger output, or machine register values.
- Keypad Input – With the debugger, you can use the numeric keypad to enter commands. The VAX/VMS Symbolic Debugger provides a default keypad layout, but you can define your own keypad bindings or alter the default bindings.
- Improved Support For Existing Languages – The debugger now understands all data types and language specific operators for each language.
- Aggregate Output – The debugger allows whole arrays and records to be examined with a single command.
- User-defined Commands – The debugger's DEFINE/COMMANDfeature, together with improvements to debugger command files and the VAX/VMS Symbolic Debugger command language, enable you to define your own debugger commands.
- Symbol Table Query – The debugger's SHOW SYMBOLcommand allows you to query the debugger about the symbols in his program.

- Improved Symbolization – The debugger's SYMBOLIZEcommand takes an address and tells you what object is at that address.
- Improved Breakpoint Facility – The debugger's capabilities for breaking, tracing, and stepping have been greatly expanded.
- SPAWN and ATTACH Commands – The debugger supports the ability to SPAWN a new subprocess much the same as is done in DCL. Then using the ATTACH command, you can move to any Debugger subprocess he has spawned or jump around between debugging sessions.
- Expandable Memory Pool – The debugger's ALLOCATEcommand allows you to allocate more space for the debugger's symbol table so you can set more (or all) modules in your program.
- Is fully supported by the VAX Language-Sensitive Editor.

▪ VAX SORT/MERGE

The VAX SORT/MERGE utility may be run interactively, as a batch job, or called from a VAX language program.

The SORT utility allows you to reorder data from one to ten input files into a single output file in a sequence based upon user-specified key fields within the input data records. If you does not wish to physically reorder the input, SORT can be used to extract key information and store the sorted information as a permanent file. That file can then be used to access the original input file in the order of the key information in the sorted file.

SORT provides four sorting techniques:

- *Record sort* produces a reordered data file by moving the entire contents of each record during the sort. A record sort can be used with any acceptable VMS input device and can process any valid RMS (Record Management Services) formatted file.
- *Tag sort* produces a reordered data file by moving only the record keys and Record File Addresses (RFAs) during the sort. SORT then randomly reaccesses the input file to create a resequenced output file according to those record keys. The tag sort method may conserve temporary storage, but can accept only input files residing on disk.

- *Address sort* produces an address file without reordering the input file. The address file contains RFAs (Record's File Addresses), a pointer to each record's location in a file that can later be used as an index to read the database in the desired sequence. Any number of address files can be created for the same database. A customer master file, for example, can be referenced by customer-number index or sales territory index for different reports. Address sort is the fastest of the four sorting methods.
- *Index sort* produces an address file containing the key field of each data record and a pointer to its location in the input file. The index file can be used to randomly access data from the original file in the desired sequence.

The MERGE utility permits you to merge data from two to ten similarly sorted input files. It merges the data according to key field(s), defined by you, and generates a single output file. All input files to be merged must be a proper subset of the equivalent SORT key fields.

The following example illustrates the sorting of a sales record file by customer last name. The name of the initial file is SALES.DAT. Each record contains six fields: date of sale, department code, salesperson, account number, customer name, and amount of sale. The numerical ranges listed below the set of records indicate the position and size of each information field within the record.

You can now rearrange the sales record in file SALES.DAT according to any of the file's information fields. For instance, to sort the file in alphabetic order of customer's last name, you would type the following command sequence:

```
$ SORT/KEY_=(POSITION:29,SIZE:30) SALES.DAT BILLING.LIS[RET]
```

In this command sequence, you are defining the SORT key to be the customer's last name and the output file to be BILLING.LIS.

You may now obtain a listing of the sorted data file by using either the TYPE or PRINT commands.

```
$TYPE BILLING.LIS[RET]
```

To perform the MERGE function, the MERGE utility expects presorted data files upon which to operate. In the following example, MERGE is operating upon two presorted (by alphabetic order) sales data files, STORE1.FIL and STORE2.FIL.

To merge the two data files, you must type the following command sequence:

```
$ MERGE/KEY=(POSITION=29,SIZE=30)STORE1.FIL,STORE2.FIL CENTR.FIL[RET]
```

You have indicated in the above command sequence that the files are to be merged via the alphabetic order of the customer's last name. You can examine the output file via the PRINT or TYPE commands.

```
$ TYPE CENTR.FIL[RET]
```

SORT/MERGE AS A SET OF CALLABLE SUBROUTINES - SORT and MERGE can be used as a set of callable subroutines from any VAX language. This subroutine package provides two functional interfaces to choose from: a file I/O interface and a record I/O interface.

▪ VAX Linker

Before a source-language program can be run on VAX/VMS, it must be assembled or compiled by a language processor and then linked. The language processors translate user-written source programs into object modules. The VAX Linker binds these object modules into an image that can be executed by the VAX system.

Not all computer systems use a linker; in some, the work of the linker is assumed by the language processors and what is called a loader. But the linker offers you greater flexibility in choosing and mixing languages, and simplifies and extends the modern approach of modular programming.

INPUT TO THE LINKER - There are two basic forms of input processed by the linker: object modules and sharable images. They are introduced to the linker as part of the input files specified in the LINK command. The linker will accept one or more of the following kinds of input files:

- Object file
- sharable image file
- Symbol table file
- Library file
- Options file

The object file can contain one or more object modules. This file has file-type OBJ. It is the fundamental input to the linker and at least one object file must be specified with any LINK command.

The sharable image file is the product of a previous linking operation, but one which is by itself not executable. It can serve only as input to another linking operation. The sharable image file can only be specified in the options file and is indicated there by the /sharable qualifier.

The symbol table file is also a product of a previous linking operation. It may be specified when linking so that the linker can use the symbol values to resolve undefined symbols in other object modules. A symbol table file has the file type STB.

There are two kinds of library files: object and sharable image. Of these there are both system libraries (maintained by VMS) and user Libraries (created by the DCL command, LIBRARY). Library files are used by the linker either to resolve undefined symbols, or as a source for particular object modules.

The options file is not really input to the linker, in the same sense the other files mentioned are input; rather, it is a tool for managing the linking operation and for simplifying the use of complex and often-repeated linker operations. (This is, in a way, analogous to the use of DCL command procedures for complex or commonly used command sequences.) A linker options file can contain one or more input file specifications, including qualifiers or special linker options that cannot be specified in the DCL LINK command line.

OUTPUT OF THE LINKER - The linker will generate one of three types of images: executable, sharable, or system, and an optional image map and/or symbol table.

The most common output of the linker is the executable image. It is the end product of program development. It has the file type EXE and can be run by the DCL command, RUN.

A sharable image, on the other hand, is not intended to be executed directly. It must be linked with one or more object modules to produce an executable image. It contains an image header, one or more image sections, and a symbol table that defines universal symbols in the sharable image.

A system image is one that does not run under the control of the operating system, but is intended to run standalone on a VAX. VAX/VMS is a system image.

If the /SYMBOL_TABLE qualifier is specified, the linker will generate a symbol table file that can serve as input to a subsequent linking operation.

ACTION OF THE LINKER - In the process of creating an image the linker performs three major tasks:

- Resolution of symbolic references
- Allocation of virtual memory
- Image initialization

The following sections describe these processes in some detail.

RESOLUTION OF SYMBOLIC REFERENCES - A symbol is a name associated with a program location or a value. Any reference to a symbol, other than the definitive reference, must be resolved. For example:

```
JMP SYMBOL_1                    (Jump to where?)
```

or

```
ADD SYMBOL_A,SYMBOL_B    (Add what to what?)
```

Somewhere, SYMBOL 1 must be defined as a location of an instruction or the beginning of a subroutine. Similarly, SYMBOL A and SYMBOL B must have had values assigned to them.

References to local symbols (that is, symbols defined and used entirely within the module) are resolved by the language processor, but references to global symbols (those that can be referred to by modules other than the defining module) and universal symbols (those referenced outside of a sharable image) must be resolved by the linker.

Since universal symbols are in fact global symbols that are available to modules outside of a sharable image, the process whereby the linker resolves global and universal symbols is the same. During its first pass through the linking operation, the linker records each symbol reference and definition in a global symbol table. When the linker seeks to resolve a symbol reference, it first searches modules named in the command line (sometimes with /INCLUDE), then user libraries, and finally system default libraries.

MEMORY ALLOCATION - By the end of its first pass, the linker has processed all the input modules and library modules needed to resolve undefined symbols, and knows how large the final image will be, but it still needs to organize the image and allocate virtual memory.

The linker organizes the image on three levels: cluster, image section, and program section.

Clusters are determined in three ways:

- The default cluster (generated by the linker)
- User-defined clusters (generated by the CLUSTERS' option)
- sharable image clusters (one for each sharable image)

Image sections are created by gathering program sections (psects) with similar attributes. Those attributes include the ability to write, execute and share, in addition to position-independence, and protected vector.

Program sections and their attributes are determined by the language and, optionally, by yourself, either through directives to the language processor (for example, .PSECT in MACRO) or by the PSECT_ATTR option in the linker options file.

The linker processes each cluster, one at a time — with the exception of non-based, or position-independent, sharable images, which are allocated virtual memory by the image activator at runtime. In processing all other clusters, the linker organizes the psects within each cluster into image sections. Then the clusters are assigned virtual address space and the image section descriptor (ISD) of each image section is updated to include the starting virtual address of the image section.

IMAGE INITIALIZATION - After resolving references and allocating memory, the linker fills in the actual contents of the image. Primarily, initialization consists of copying all data and code into a single image; but the linker performs two other functions at this stage: it computes values that depend on externally defined fields, and it inserts these values into the referencing location.

FIX-UP IMAGE SECTION - After it has initialized the image, the linker will generate a special image section, called the fix-up image section. This image section contains the code that makes otherwise position-dependent sharable images position independent.

The general addressing mode is used to reference routines and data contained in a sharable image. The linker converts general addressing mode directives into longword-deferred addressing mode, with indirection going through the fix-up image section. Failure to use general addressing mode when referencing a sharable image will elicit a warning message.

All DIGITAL VAX high-level languages generate position-independent code.

SHARABLE IMAGES - An important benefit of the linker is that it allows the use of sharable images. An effective application of sharable images can help to conserve valuable resources in the your operating environment. For example, physical memory requirements would be reduced if global sections (one for each image section of a sharable image) used commonly among processes could be resident in memory and mapped into their address space. Thus the same physical pages satisfy a number of processes, reducing duplication. So, too, you can conserve disk storage and reduce paging I/O, when sharing replaces duplication.

One of the reasons modular programming is so attractive is that a commonly used routine or function can be developed or modified once, then incorporated into any number of programs. The use of sharable images carries this efficient practice a step further. The modules that make up a sharable image are linked only once, so the overhead of resolving undefined symbols (within the image) and generating image sections — the bulk of the linker's work — is incurred only once, facilitating another level of modular hierarchy. Furthermore, since a position-independent sharable image is allocated to virtual memory by the image activator at runtime, the code it includes can be modified and updated without having to relink every program that uses that image.

THE LINK COMMAND - The linker is run by the DCL command:

```
$ LINK  [/Command-qualifier...] file-spec [/file-qualifier...]...
```

At least one input file must be specified. There can be multiple command qualifiers, multiple file specifications, and multiple qualifiers for each file specified.

▪ The VAX Text Processing Utility (VAXTPU)

VAXTPU is a high-performance programmable text processing utility available in VMS. VAXTPU is a tool designed to aid application and system programmers in the development of text processing interfaces. The utility includes a compiler, an interpreter, a high-level procedural language, and two editing interfaces written in VAXTPU.

VAXTPU Interfaces

You can tailor one of the existing VAXTPU editing interfaces to suit your editing style or you can write your own editing interface with VAXTPU. You can use VAXTPU to design an intelligent editor for a specific environment. The editor or other application that you layer on top of VAXTPU becomes the interface between you and VAXTPU. You must use one of the existing VAXTPU interfaces or create your own interface in order to access VAXTPU.

You can think of VAXTPU as a base on which text processing interfaces can be layered. The two editing interfaces included in the VAXTPU kit, the Extensible VAX Editor (EVE) and the VAXTPU EDT Keypad Emulator, are good examples of interfaces that are written in VAXTPU and layered on VAXTPU. See Figure 5-2.

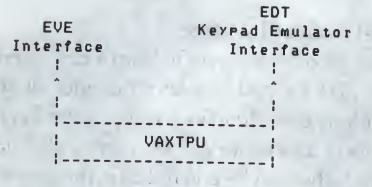


Figure 5-2 ▪ VAXTPU As a Base for EVE and the EDT Keypad Emulator

You can write extensions for the two interfaces that are shipped with VAXTPU or you can write a completely separate interface for VAXTPU. See Figure 5-3.

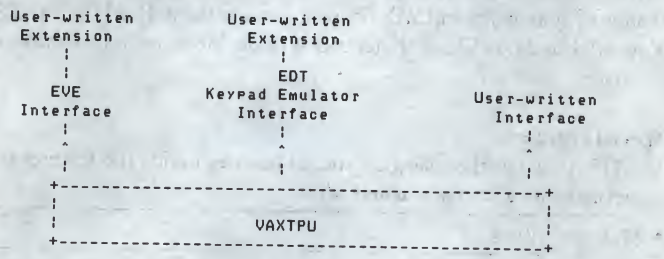


Figure 5-3 ▪ VAXTPU As a Base for User-written Interfaces

Extensions to an existing interface can be implemented with a VAXTPU command file (VAXTPU source code) or with a VAXTPU section file (compiled VAXTPU code in binary form). Because a VAXTPU section file is already compiled, start-up time for your interface will be shorter when using a section file as opposed to a command file.

The only way to implement an interface that is entirely user-written is with a section file.

The EVE Interface

The Extensible VAX Editor (EVE) is a new editing interface that is easy to learn and fast to use. Users with little or no experience with text editors can quickly learn to perform basic editing tasks with the EVE interface. The most common editing functions are accessed by pressing a single key on the EVE keypad.

EVE is also a powerful and efficient editor for experienced users of text editors. The more advanced editing functions are accessible by entering commands on the EVE command line. Many of the special features of VAXTPU (such as multiple windows) are available with EVE commands. Read the User's Guide to EVE to learn more about the EVE interface.

The EDT Keypad-Emulator Interface

For those EDT users who do not want to learn a new interface for basic editing tasks, the VAXTPU EDT Keypad Emulator provides all of the functions of the EDT keypad and binds these functions to the same keys that EDT uses. The EDT Keypad Emulator also provides EDT users with a limited set of the EDT line mode commands that can be entered after the asterisk that appears when you press CTRL/Z.

The EDT Keypad Emulator provides access to advanced VAXTPU functions on the VAXTPU command line. To access the VAXTPU command line, press PF1/7. When you see the prompt, "VAXTPU command: ", you can enter VAXTPU commands or statements and VAXTPU will execute them. Read the VAXTPU EDT Keypad Emulator Quick Reference Guide for more information on this interface.

Special Features

VAXTPU provides the following special features beside the features normally associated with a screen-oriented editor:

- Multiple buffers
- Multiple windowing
- Multiple subprocesses within VAXTPU and at DCL level
- Text processing in batch mode
- Insert or overstrike text entry
- Free or bound cursor motion
- Learn sequences
- Pattern matching
- Key definition
- Procedural language
- Callable interface

If you use the EVE interface to VAXTPU, many of the special features of VAXTPU are easily accessible with a single EVE command. If you use the EDT Keypad Emulator interface, you must include a user-written extension to the interface to have easy access to these special features.

Hardware and Terminals That VAXTPU Supports

Because VAXTPU does not use a work file, but does all of its work in memory, you may have to adjust some system parameters or divide files into smaller segments if the files you want to work with are very large.

VAXTPU supports screen-oriented editing on all DIGITAL video display terminals, except the VT52. One of the major goals in the design of VAXTPU was fast performance for screen-oriented editing. Optimum screen-oriented editing performance occurs when running VAXTPU from VT220 and VT100 terminals. Some Digital video display terminals have hardware behavior that does not allow optimum VAXTPU performance.

Although you cannot use the VAXTPU screen-oriented features on the VT52 series of terminals, on hardcopy terminals, or on foreign terminals, you can run VAXTPU on these terminals if you use a line mode style of editing.

The VAXTPU Language

VAXTPU is a high-level procedural programming language that allows you to perform powerful text processing tasks. The VAXTPU language can be viewed as the most basic component of VAXTPU. In order to access the features of VAXTPU, write a program in the VAXTPU language and then use the VAXTPU utility to compile and execute the program. A program written in VAXTPU can be as simple as a single VAXTPU statement, or as complex as the section file that creates the EVE interface.

The VAXTPU language is block-structured and is easy to learn and use. VAXTPU language features include Boolean operators, error interception statements, looping, case, and conditional statements, and an extensive set of data types. Comments are indicated with a single comment character, so that you can document your procedures internally. There are also capabilities for debugging procedures with user-written debugging programs.

VAXTPU Data Types

The VAXTPU language has an extensive set of data types. Data types are used to interpret the meaning of the contents of a variable. Unlike many languages, the VAXTPU language has no declarative statement to enforce which data type must be assigned to a variable. Variables in VAXTPU assume a data type when they are used in an assignment statement.

The following statement assigns a string data type to the variable this_var:

```
this_var := 'This can be a string of your choice.'
```

The following statement assigns a WINDOW data type to the variable X. The window occupies 15 lines on the screen, starting at line 1 and the status line is OFF (not displayed):

```
X := CREATE_WINDOW (1, 15, OFF)
```

Many of the VAXTPU data types (for example, LEARN and PATTERN) are different than data types usually found in programming languages. Following is a list of VAXTPU data types:

- NULL – The initial state of a variable after it has been compiled, or added to the VAXTPU symbol table.
- STRING – A string constant.
- INTEGER – An integer constant.
- BUFFER – A collection of text records. The CREATE BUFFER procedure returns this data type as its result.
- WINDOW – A Window data type is a subdivision of the screen. The CREATE WINDOW procedure returns this data type as its result.
- MARKER – A position within a buffer, tied to the character that resides at that buffer position. The MARK procedure returns this data type as its result.
- RANGE – All of the text that occurs between and including two markers. The CREATE RANGE procedure returns this data type as its result.
- PATTERN – A collection of pattern expressions. The pattern operators and the pattern built-in procedures return a pattern as a result.
- PROGRAM – A collection of VAXTPU executable statements. The COMPILE procedure returns this data type as its result.
- PROCESS – A VMS subprocess. The CREATE PROCESS procedure returns this data type as its result.
- LEARN – A collection of VAXTPU keystrokes. The LEARN END procedure returns this data type as its result.

VAXTPU Language Statements

VAXTPU language statements include the following:

- An assignment statement (:)
- Procedure statements (PROCEDURE–ENDPROCEDURE)
- Repetitive statements (LOOP–EXITIF–ENDLOOP)
- Conditional statements (IF–THEN–ELSE–ENDIF)
- Case statements (CASE–ENDCASE)
- Error statements (ON ERROR–ENDON ERROR)

VAXTPU Built-in Procedures

The VAXTPU language has many built-in procedures that perform functions such as, screen management, key definition, text manipulation, and program execution.

You can use built-in procedures, for example, to create an editing interface with multiple windows. Multiple windows allow you to see parts of different files (or different parts of the same file) on your screen at the same time. The CREATEWINDOW procedure allows you to create and alter the number, position, and size of the windows on the screen. Since you can also create multiple buffers and associate them with windows on the screen, you can edit multiple files concurrently with VAXTPU.

VAXTPU allows you to run more than one process concurrently. You can use built-in procedures to create multiple processes. The CREATEPROCESS built-in procedure allows you to create a subprocess and attach a buffer to it for storing the output of the subprocess. In one process you can compile a large program, sending any error messages to a message file, while you continue to develop a program in another process. Limitations on subprocess activity are the same as the limitations for VAX/VMS subprocesses.

You can use the DEFINEKEY procedure to bind a VAXTPU program to a particular key, or key combination. This allows you to control the actions that are taken when a key is pressed. You can use the SAVE built-in to keep your key definitions from session to session.

You can use the UNDEFINEKEY procedure to remove a key binding. The following example is a built-in procedure that removes the association between the key sequence CTRL/Z and the code that it previously executed:

```
UNDEFINE_KEY (CTRL_Z_KEY)
```

You can use built-in procedures as statements within a VAXTPU procedure, or as commands from either the EVE interface or the EDT Keypad Emulator interface. See the interface manuals for a description of how to enter VAXTPU procedures or commands.

VAXTPU User-written Procedures

You can write your own procedures that combine VAXTPU language statements with calls to VAXTPU built-in procedures. After you compile a procedure, you can save it and call it when you need it.

The body of a procedure can contain zero or more VAXTPU statements. Statements are separated by semicolons. VAXTPU procedures can return values. Procedures can be recursive.

Example 5-1 is a sample procedure that uses VAXTPU language statements (PROCEDURE – ENDPROCEDURE) and built-in procedures (SET and MOVE_VERTICAL) to slow down the action of the up arrow key:

```
! This procedure slows down the speed
! of the up arrow key while scrolling

PROCEDURE user_slow_up_arrow

        SET (AUTO_REPEAT, OFF);
        MOVE_VERTICAL (-1);
        SET (AUTO_REPEAT, ON);

ENDPROCEDURE
```

Example 5-1 ▪ Sample User-written Procedure

Invoking VAXTPU

To invoke VAXTPU at DCL level, simply type EDIT/VAXTPU, followed by the name of your file. For example:

```
$ EDIT/VAXTPU text_file.lis
```

This command opens the file text_file.lis for editing. Note that you can specify only one input file on the command line. You can include additional files from within VAXTPU later in your editing session.

When you invoke VAXTPU with the preceding command, you are placed in a default editing interface (unless you specify /NOSECTION). The default editing interface for VAXTPU is EVE. We suggest that you create a symbol like the following so that you can invoke EVE with a command that suggests the interface being used:

```
$ EVE :==EDIT/VAXTPU
```

If you want to invoke VAXTPU with the EDT Keypad Emulator rather than EVE as your editing interface, enter the following command after the system prompt:

```
$ EDIT/VAXTPU/SECTION=EDTSECINI.GBL
```

Instead of typing such a long command line, you can create a DCLsymbol like the following to invoke VAXTPU with the EDT Keypad Emulator interface:

```
$ EDTEM :== EDIT/VAXTPU/SECTION=EDTSECINI.GBL
```

Then you can simply type EDTEM at DCL level to invoke VAXTPU with the EDT Keypad Emulator interface. The terminal screen will look exactly as it does when you use EDT, except for the two lines reserved at the bottom of the screen for error and informational messages from VAXTPU.

EDIT/VAXTPU Command Qualifiers

You can add qualifiers to the DCL command EDIT/VAXTPU. VAXTPU qualifiers control such items as recovery and initialization files. Qualifiers to the EDIT/VAXTPU are listed in Table 5-1.

Table 5-1 ▪ Qualifiers to the DCL command EDIT/TPU

Qualifier	Default
/[NO]COMMAND	/NOCOMMAND
/[NO]SECTION	/SECTION = EVESECINI
/[NO]DISPLAY	/DISPLAY
/[NO]OUTPUT	/OUTPUT
/[NO]JOURNAL	/JOURNAL
/[NO]RECOVER	/NORECOVER
/[NO]READ_ONLY	/NOREAD_ONLY

Initialization Files

You can use two kinds of initialization files to create or customize a VAXTPU interface: command files and section files.

A command file is a VAXTPU source code file that has a file type VAXTPU. It is used with the VAXTPU qualifier /COMMANDfile-spec. By default, no command file is read when you invoke VAXTPU. You must specify /COMMANDfile-spec if you want to include a command file.

A section file is a compiled VAXTPU file. It is a binary file that has a GBL file type. It is used with the VAXTPU qualifier /SECTIONfile-spec. By default, the section file that creates the EVE interface is read when you invoke VAXTPU. You must specify a different section file (for example, /SECTIONmysectionfile) or /NOSECTION if you do not want to use the EVE interface. Note that when you specify /NOSECTION, there is no interface to VAXTPU. Even the RETURN and DELETE keys are not defined. /NOSECTION is used in the process of creating your own section file.

You can use either a command file or a section file to customize or extend an existing interface. A command file is generally used for minor customization of an interface. A section file is used to customize or extend an interface when the customization is lengthy or complex because start-up time is faster with a section file.

To create an interface that is not layered on top of one of the existing interfaces, use a section file.

Copies of both the command file and the section file for EVE and the EDT Keypad Emulator are in SYS$SHARE. We included the command files on-line so that you can read the VAXTPU source code to see how VAXTPU was used to create the two different editing interfaces. The VAXTPU source file for EVE is SYS$SHARE:EVESECINI.TPU. The compiled binary section file for EVE is SYS$SHARE:EVESECINI.GBL. The VAXTPU source file for the EDT Keypad Emulator is SYS$SHARE:EDTSECINI.TPU. The compiled binary section file for the EDT Keypad Emulator is SYS$SHARE:EDTSECINI.GBL. If you cannot find these files on your system, see your system manager.

Leaving a VAXTPU Editing Session

When you want to leave a VAXTPU editing session, you can either QUIT or EXIT. If you QUIT the EVE or the EDT Keypad Emulator editing interfaces, the work that you have done will not be saved. If you EXIT from these interfaces, the current buffer will be written to a disk (if it was modified) and you will be queried about writing other modified buffers to a disk.

▪ The EDT Editor

EDT, the Digital standard editor, lets you enter and manipulate text and programs. With its extensive HELP facility, the EDT editor is designed to be learned by novices. In addition, it provides many capabilities that will appeal to advanced users.

WHAT EDT DOES - EDT is a interactive text editor that provides:

- Both line and character editing facilities
- Screen editing using the keypad on VT100 or VT200 series video terminals
- The ability to work on multiple files simultaneously
- A journaling facility that protects against loss of edits due to system crashes
- An extensive HELP facility
- A default start-up command file, which allows a choice of editing options to be set automatically
- A window into a file (on VT100 and VT200 terminals only) that lets you view changes in buffer contents immediately
- Sharable installation for many users

BUFFERS - All editing with EDT is done using buffers. A buffer is a part of EDT's memory. When you begin editing, the input file is read into the MAIN buffer, and when editing is complete, the MAIN buffer is written onto the disk as a file. Thus, editing in the MAIN buffer is like editing a file directly.

START UP FILE - When you invoke EDT, the editor checks to see if you created a start-up file. Editing options, such as SET MODE CHANGE and DEFINE KEY, can be inserted in the start-up file. These options take effect automatically when an editing session begins.

HELP FACILITIES - The HELP facilities on EDT are extensive. You can get help on general EDT operations by typing HELP. If help is needed while in keypad mode, pressing the help key displays information that is specific to keypad editing. The help information is tree-structured, so that more specific help can be obtained on a general topic.

REDEFINING KEYPAD KEYS - You can redefine any of the keypad keys and most of the control (CTRL) keys on VT100 or VT200 terminals. With this feature, a series of commands can be assigned to a key. EDT then performs these commands when the key is pressed.

THE SET AND SHOW COMMANDS - The SET command, with a variety of qualifiers, affects EDT's editing capabilities. SET controls screen parameters such as line width. SET also lets you determine the appearance of text, such as changing the window size to less that 22 lines. The SHOW command provides information on the current state of the editor, such as terminal parameters, definitions of keypad keys, and the names of buffers in use during an editing session.

JOURNAL PROCESSING - Journal processing protects your work against system crashes. During an editing session, EDT saves all the input from a terminal in a journal file. After a system crash and restart, you can retrieve and execute commands in this saved file with the /RECOVER option. In this way, an editing session can be recovered to nearly the time of the crash.

EDT MODES OF OPERATION - With EDT there is a choice of keypad or line mode editing. They allow you to

- Display a range of lines.
- Find, substitute, insert, and delete text.
- Move, copy, and renumber lines.
- Copy text into a buffer and write it on files.
- Define the functions of keys.

Keypad editing is available on VT100 and VT200-series terminals. The group of keys at the right of the keyboard is used to enter keypad functions.

Keypad editing is powerful and versatile, yet it is easy to learn and use. In keypad editing, the active buffer is displayed on the screen as you edit. There is a variety of editing functions available, each of which requires that only one or two keypad keys be pressed to perform a function. You enter commands, inserts text, and performs CONTROL functions from the keyboard.

▪ Document-Formatting Utility (DSR)

Designing and producing printed materials can be simplified through the use of the Digital Standard RUNOFF (DSR) utility. DSR reduces the time needed to prepare a document by allowing both textual corrections and formatting changes to be executed in the same pass over the file. And since text changes do not affect the basic design, documents can be updated without extensive retyping.

The input to DSR is a file containing the text of the document and the DSR instructions. The output file is the print-ready document. After the program has been run, the original file remains available for further editing.

Formatting instructions consist of commands and flags. Command lines are signaled by a command flag, usually a period, in position one and can contain one or more commands and text. Within the text are special characters — called flags — that specify character enhancements such as underlined text or boldface characters.

FILLING AND JUSTIFYING - DSR commands can set left and right margins, so that you can enter text without concern for line width or variable spacing between words. The DSR program can fill and justify the text when it is run. Filling is the successive addition of words to a line until one more word would exceed the right margin. DSRjustifies the line by adding enough spaces between words to expand the line to the right margin.

DSR DEFAULT MODES - When an input file is processed by the Digital Standard RUNOFF utility, certain default actions are performed that do not depend upon command or flag entries for their execution. These actions are similar to those performed during the preparation of a manually typed document.

DSR default modes provide

- A standard page size, 70 characters x 58 lines.
- Sequential page numbering.
- Right margin of 70 characters, left margin 0.
- Single spacing.
- Automatic tab settings for every eight print positions.
- Automatic filling and justifying is turned on.

PAGE FORMATTING - The page formatting commands control the appearance of each page for output. For example, there are page formatting commands to enable or suspend page numbering, produce and format titles and subtitles, or force the printer to advance to a new page.

Another page formatting command allows a conditional page advance, based on the number of lines left on the page. This capability can be used to guarantee that text which should appear on a single page (for example, tables and lists) will not be broken up.

For example:

LAYOUT 2,5

The 2 indicates page titles will be flush right on odd pages, flush left on even pages; pages will be numbered at center bottom with 4 blank lines after the body of the text.

TITLE FORMATTING - Title formatting commands provide page, title, and subtitle information for all pages. Such actions as placing only the chapter heading on the first page of a chapter and printing any subtitles are provided for by the title formatting commands.

SUBJECT-MATTER FORMATTING - Subject-matter formatting includes commands for managing the design and appearance of text, such as making a ragged right margin, indenting a paragraph, skipping a number of lines, centering a line of text, underlining, hyphenating, and overstriking. Parts of the text can be formatted differently from one another, and commands can be combined. For example, you can have lists justified or having them with ragged margins.

Table 5-2 ▪ Selected Examples of DSR Subject-Matter Formatting Commands

Command Sequence	Result
.LM 5.RM 58	Set the left margin at space 5 and the right margin at space 58
.NF	Disables filling: causes a new line in the input file to produce a new line in the output file
.NJ	Disables justifying, lines are ragged right
.BR	Causes a break: current line is output without being filled or justified
.S or .SK 2	Skips two blank lines
.PG	Causes a .BR then starts the next page
.TP 25	Tests the current page to determine if there is a minimum of 25 lines remaining. This is done so certain segments of text (i.e., paragraphs, or listings) can be kept together and not create "widows at the beginning of a page.
.center	Centers subsequent line of text between the left and right margin.
.TS 3,7,19,15,26...	Sets up to 32 new tap stops to override the default tab stop values
.P 4,2,3	Formats paragraphs in which: first word is indented 4 spaces; there are 2 blank lines between paragraphs; there must be at least 3 lines remaining on the page for the paragraph to be started on the current page.

GRAPHIC, LIST, AND NOTE FORMATTING - It often becomes necessary to accommodate graphics, lists, and tables, or to allow for special notes to be inserted. Footnotes also have to be prepared in such a way as to fit on the appropriate pages of the final document.

Table 5-3 ▪ DSR Graphic, List,and Note Formatting Examples

Command	Result
.FIG 24	Leaves 24 lines for a figure to be inserted
.FIG DEF 30	Leaves 30 lines, including at the top of the next page, for a figure
.LIST 1, "*"	Sets up a list with 1 blank line between items and an asterisk marking each item
.LE	Identifies the start of an element
.DLE "(",LL,")"	Establishes a user-specified display format for lists: in this case, sequential, lowercase letters will be enclosed in parentheses.

These commands provide a properly numbered and formatted outline. The right column indicates their output if these three header levels appear in chapter 14 of a publication.

.HL 1 Plays	14.1 Plays
.HL 2 King Lear	14.1.1 King Lear
.HL 3 Tragic Flaw	14.1.1.1 Tragic Flaw

MISCELLANEOUS FORMATTING - Several useful DSR commands help you to add comments (not printed to the output file) to the source file, to gather externally located files into the input, to exert conditional control, and to set or display time and date.

Table 5-4 ▪ Miscellaneous Formatting Commands

Command	Example
.IF complete	Processes the line following only if the qualifier /VARIANT:COMPLETE was given on the command line.
!appendix C	DSR ignores comments
.ELSE complete	Marks the end of the line to process because of the IF, and starts the alternative
.ENDIF complete	Marks the end of a group of conditionally processed lines using the variant "complete"

FLAGS - Flags are special characters (for example, an ampersand) that perform specific operations (for example, underlining). The specified operation is invoked when a character is recognized as a flag by DSR. Certain special characters initially are recognized by default.

INDEX AND TABLE OF CONTENTS - DSR has powerful facilities for creating indexes and tables of contents easily. The TOC program generates tables of contents.

Table 5-5 ▪ DSR Flag, Index, and Table of Contents Commands

Flag, Index, and Table of Contents Commands	
fix#some#space	The SPACE flag (#) fixes one nonexpandable space whenever it occurs
R_&D	The ACCEPT flag (_) prevents DSR from interpreting the ampersand in R&D as an underline flag
.X Satire	Creates an index entry for Satire. DSR gives it the current page number
.ENTRY Parody>See Satire	Provides a cross reference to the index using the subindex flag

RUNNING THE DSR PROGRAM - DSR is initialized by entering the following command:

```
RUNOFF filespec↵
```

After processing the file, DSR terminates.

Table 5-6 ▪ DSR Run Commands

$RUNOFF MYBOOK	Processes MYBOOK.RNO and produces MYBOOK.MEM as output

Various qualifiers can be placed on the command line. Examples are:

/FORMSIZE=55	Sets page to 55 lines rather than the default of 60 lines
/PAGES=3-1:3-16, 4-1:4-16"	Prints only pages 3-1 through 3-16 and 4-1 through 4-16
/DEBUG:echo	Traces the operation of any DSR commands defined by a parameter by echoing each execution in the output file
/OUTPUT:TT:	Directs output to the terminal

▪ Other Program Development Utilities

Some of the other major program development utilities available to you are:

- The Command Definition Utility
- The Object Analyzer Utility
- The Message Utility
- The Exchange Utility

The Command Definition Utility (CDU)

The VMS Command Definition Utility (CDU) creates, deletes, or changes command definitions in a command table. As input, the CDU accepts a command table and/or a file that contains command definitions. The CDU processes this input to create a new command table. The new table can be either executable code or an object module.

The CDU provides a way of defining command line syntax. The command table is used by the CLI (command line interpreter) to parse commands. The CLI is callable from the VAX Common Language Environment.

The OBJECT Analyzer Utility.
The object module analysis utility checks an object module (or a concatenated file containing several object modules) to see if it is in the correct format for input to the linker. It is a diagnostic tool for writers of compilers or assemblers that generate VAX object code. The program, invoked by the DIGITAL Command Language (DCL) command ANALYZE/OBJECT, can analyze the entire module or only specified types of records. It checks the record type, contents, and sequence of each object module record it examines. The program creates an output file containing a record-by-record analysis of the object module, including identification of any errors in the module.

The MESSAGE Utility
The MESSAGE utility allows you to construct informational, warning, or error messages in standard VAX/VMS format. First, using a text editor, you create a source file that specifies the information used in messages, message codes, and message symbols. The MESSAGE command can then be used to compile the source file.

The EXCHANGE Utility (EXCHANGE).
The VMS Utility EXCHANGE allows you to transfer files between foreign volumes and VAX/VMS native volumes. It appropriately converts the file format when transferring files between different structured volumes. EXCHANGE recognizes all Files-11 volumes of VAX/VMS devices, as well as all DOS-11 formatted volumes on nine-track magnetic tape devices.

EXCHANGE also recognizes several foreign file structures and supports most operations that are useful for each volume structure. It allows you to initialize and manipulate functions of the foreign volumes, for example.

You can use the EXCHANGE Utility to

- Locate bad blocks on volumes.
- List directories of volumes.
- Transfer files to and from volumes.
- Delete files and compress volumes for block-addressable devices (such as RT-11 disks.)

The EXCHANGE Utility employs defaults to ensure volume formats and file structures are compatible with the type of operation you want to perform.

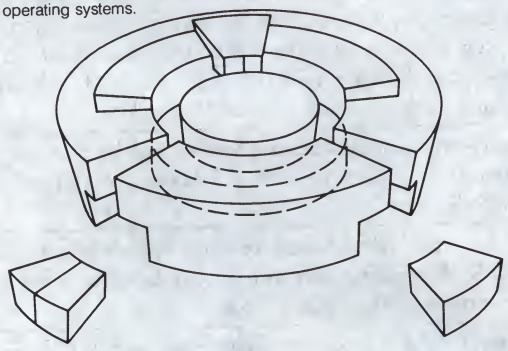
Chapter 6 ▪ VAX Program Migration and Cross-Development Tools

▪ Overview

One important benefit inherent in VAX/VMS systems is Digital's commitment to software compatibility among the entire family of VAX processors and, in many cases, other Digital systems as well. As a VAX/VMS user, you can migrate, or move, many existing applications from one system to another and even develop application programs on a VAX that will be executed on other Digital target systems. These capabilities are realized through the use of the VAXELN Toolkit, VAX-11 RSX, and the MicroPower/Pascal-VMS Toolkit.

- Realtime and other statically defined systems can be developed on a VAX/VMS system with the VAXELN Toolkit and then run on a target VAX processor without VMS present.
- Migration and cross development of many applications to and from the RSX-11 family of operating systems is accomplished with the optional program development product VAX-11 RSX.
- MicroPower/Pascal is a modular executive and software development tool that runs under VMS and is used to develop PDP-11 (Q-Bus) based microcomputer applications.

This chapter introduces you to 3 VAX/VMS optional software products that provide for the migration and cross development of application programs between the VMS operating system and other Digital operating systems.



VAX-11 RSX
MicroPower/Pascal-VMS

VAX-11 RSX simulates the RSX-11M/11M-PLUS operating system environment by providing

- PDP-11 users with a familiar user interface to the VMS operating system.
- Many RSX-11 and VMS program development utilities and other facilities for writing RSX-11 programs that will execute in VMS compatibility mode or under the RSX-11 operating system.
- An Applications Migration Executive (AME) that provides a means of executing most nonpriviledged RSX-11 programs on VMS.

The VAXELN Toolkit

The VAXELN Toolkit supports the development of stand-alone, statically defined software systems. VAXELN applications

- Are developed under VMS with VAXELN Toolkit components and VMS program development facilities, used for standard VMS program development.
- Applications runs on the entire line of VAX processors without the VMS operating system present.
- Are typically, although not necessarily, used to develop realtime applications.

Figure 6-1 ▪ Overview Of Chapter 6

▪ VAXELN Toolkit Overview

The VAXELN Toolkit is an optional VAX software product for the development of dedicated, realtime VAXELN systems that run on VAX and MicroVAX computers. The toolkit runs on any VAX processor running the VMS or MicroVMS operating system. This system is call the *host system* . Once you have finished building the application, called the VAXELN system, it runs directly on a *target VAX processor* without any operating system present.

Typical VAXELN applications run on individual processors used for "dedicated" or otherwise predetermined functions and that are not needed simultaneously for general computing, program development, or other uses for which a general operating system, VMS for example, is more appropriate. Examples of situations requiring dedicated applications are industrial automation, workstations designed for a particular profession, Ethernet server networks, or robots.

VAXELN is especially suited to, although not limited to, creating realtime applications. These software programs are used by a processor operating in an environment where response to external events is critical. Such applications include the typical scientific and industrial data processing situations in which the computer's operation has to be precisely synchronized with machines and special input/output devices.

The VAXELN toolkit simplifies the design and implementation of such applications by offering high-level implementation languages (Pascal and C), a conceptually simple and small kernel executive (which manages resources, processes, and data), and pregenerated optionally included service programs and device drivers (which implement a file system, network communication facilities, and I/O device handling).

VAXELN provides multitasking in Pascal or C programs. Therefore, you can write a program made up of several concurrently executing parts. Besides multitasking, multiprogramming is also supported, This means that entire programs, including multitasking programs, can be scheduled concurrently on the same CPU. (Multiprogramming is different from multiprocessing. Multiprocessing programs literally execute in parallel, on different processors on the same bus. Multiprocessing is not available through VAXELN.)

VAXELN Systems

A VAXELN system is a set of programs executing on VAX hardware, along with standard code and data that manage the program's execution. VAXELN systems can run on individual VAX or MicroVAX computers or, with networking software provided in the Toolkit, they can be connected in an Ethernet local area network (LAN) This network may include VAX/VMS nodes or any other nodes using the DIGITAL Network Architecture DECnet services and protocols.

Since DECnet is supported by all of DIGITAL's operating systems, VAXELN applications can communicate with programs running on processors anywhere in a DECnet network. This makes it easy to distribute an application's programs among several network nodes, and changing the network location of a program typically requires no changes to the program code.

The hardware configuration for a VAXELN system also includes optional peripheral devices, such as disks and terminals, and communication hardware to support the execution of the programs on various nodes in a LAN. Besides, the configuration may include special hardware you have designed or acquired, such as custom device interfaces.

The programs executing in a VAXELN system are of two kinds:

- Your programs. These can include user-written device drivers or resource services, as well as typical computational programs.
- Programs (services and drivers) supplied by DIGITAL. Examples are the Network Service, the File Service, and drivers for the standard supported peripheral devices.

You can develop VAXELN systems entirely in a high-level language, including the handling of devices, exceptions, timeouts, and power failures. The recommended languages are VAXELN Pascal or the VAX C programming language.

VAXELN Pascal is a compatible superset of ISO-standard Pascal. Any program written in ISO-standard Pascal can be compiled by the VAXELN Pascal compiler and executed as part of a VAXELN system. VAXELN Pascal is supported by a highly optimizing compiler that generates position-independent, native-mode code. It is the primary implementation language of the VAXELN Toolkit itself.

VAXELN also supplies you with a set of C runtime library functions that provides you with the capability to write C programs running under VAXELN. The VAXELN C Runtime library contains a compatible subset of the VAX/VMS C Runtime library and the typical UNIX* runtime environment. It also provides access to all VAXELN features.

Many C programs originally written for VAX/VMS or UNIX will run in a VAXELN system with only minor modifications. However, the VAXELN C runtime library does not support all VAX C or UNIX-emulation functions.

*UNIX is a Trademark of AT&T.

You develop a VAXELN system by writing new programs in VAXELN Pascal or VAX C. Programs can also contain existing subroutines written in other VAX languages, provided that they do not call VAX/VMS services or language-specific runtime routines calling VMS services. Then, with simple VAX/VMS commands, you combine the programs with each other, with any of the standard services and drivers you want, and with the VAXELN kernel to form an executable system. If you are programming for a set of computers linked by a network, you simply prepare a VAXELN system for each connected machine, or node.

Once a VAXELN system has been prepared, the system image is ready to be booted on a target processor. A VAXELN system can be booted from a disk, from a TU58 tape cartridge, from a diskette, or, if the host system has the optional DECnet-VAX license and Ethernet hardware, by down-line loading the system into the target computer. VAXELN system images also are suitable for placement in read-only memories (ROMs) and booting from them. However, the ROM-blasting equipment and software are not part of the Toolkit and must be acquired separately.

After booting or downline loading a VAXELN system image onto each target machine in your local area network, you have a completely defined VAXELN application. The typical structure of such a network-based application consists of:

- A VAX processor running the VAX/VMS or MicroVMS operating system that serves as the host development system. This processor is used to develop and build each VAXELN system. It also contains the VAXELN debugger, which can remotely access one or more VAXELN target system nodes at the same time for debugging purposes.
- One or more target machines connected by the Ethernet to the VAX processor serving as the host development system and to each of the other target machines. Each target machine is a node in the network and contains its own running VAXELN system.

Toolkit Components

Along with other software modules, you receive the following program development utilities (VAX/VMS program images) in the VAXELN Toolkit.

- The VAXELN Pascal compiler.
- The VAXELN debugger.
- The VAXELN system builder.

Application programs are written with the aid of the usual VMS text editors and other utilities and are compiled with the VAXELN Pascal compiler or the VAX C compiler (acquired with a separate license), as appropriate to the development language in use. The compiled code is linked to special runtime libraries also supplied with the Toolkit, using the standard VMS linker.

The runtime libraries provide special support for VAXELN Pascal and VAX C I/O operations, the standard Pascal routines such as SIN, the standard C routines commonly associated with UNIX such as printf, and certain procedures used in system programming. The libraries are provided both in object-library and shareable-image forms in the Toolkit. In the latter case, only those shareable images containing code called by application programs are included in the finished VAXELN system, resulting in a finished system with a minimal amount of unused code, while maintaining maximum ease of use in program development.

The VAXELN Pascal compiler can also be used for VMS programming. The compiler generates the same Debugger Symbol Table (DST) information as used by the VAX Symbolic Debugger. It is, therefore, possible to use that debugger to debug VAXELN Pascal programs running under VMS.

The VAXELN debugger is used to debug the programs in a developed, executing VAXELN system. It can be used to debug a VAXELN system "locally" using the target computer's console terminal, or, if you have the optional DECnet-VAX license and Ethernet hardware, it can be used remotely to debug VAXELN systems running on Ethernet nodes from your terminal on a VAX/VMS node.

In the remote case, you can examine and manipulate variables and other items by their declared names (symbolic debugging) and can examine lines of source code in the program being debugged. The remote debugger can display the states of all VAXELN processes and jobs in the local area network, and it can dynamically change your "session scope" from one process or node to another.

In either case, when using the Debugger, you can evaluate expressions, execute a large set of debugger commands in the same general syntax as used in the VAX debugger, define new debugger commands and variables for use in a debugging session, and debug kernel-mode code.

The VAXELN System Builder combines program images, the VAXELN kernel image, and run-time routines to produce an image of the finished VAXELN system.

Also included in the Toolkit are a number of program images ready for inclusion in your VAXELN system (additional information is given later in this document).

The VAXELN File Service supports I/O operations from VAXELN programs to file-storage devices, as well as remote file access to and from other DECnet nodes. I/O requests from your programs are interpreted by the File Service and performed by the appropriate device driver program. The File Service and the Toolkit's disk driver programs use the DIGITAL Data Access Protocol (DAP), Version 7.0, for all low-level I/O operations with consumer programs and remote requests. Any user-written device drivers can be combined with the File Service and programming tools are supplied in the Toolkit for this purpose.

In a network application involving several VAXELN nodes, only one needs to provide the File Service (and disk or tape hardware and driver) for use by all the others. That is, the node can act as a file server for the others.

The VAXELN Network Service provides completely transparent network communication between VAXELN nodes in a local area network and between VAXELN nodes and other DECnet nodes. VAXELN provides the capability to restrict node access to a specified list of users and for a program to determine the identity of a user issuing a network request through an optional service called the Authorization Service (explained later in this document).

In network applications, each VAXELN node runs its own VAXELN system, and each system is built including the Network Service. Given such a configuration, the network locations of VAXELN application programs are completely invisible to each other; that is, a program can communicate with a program on another node using precisely the same statements as if both programs were on the same node. Internode communication is transparent.

The VAXELN kernel is included in every VAXELN system. It manages the system's processes and data, providing the controlled sharing of the system's resources. The operations of the kernel are reflected in VAXELN programs by special procedure calls, almost all of which are predeclared in the language. (A few kernel procedures are not predeclared, although their calling interfaces are documented and are provided in declarations that you can include explicitly. These procedures are low-level routines and are typically of no use to general programming.) For C programming, these kernel procedures are contained in a include module.

High-performance device drivers are supplied for the commonly used UNIBUS (VAX) and Q-22 bit (MicroVAX) devices. All are implemented in VAXELN Pascal and are supplied both in source form and in image (binary) form. The driver sources can be used as templates for user-written drivers. (Refer to the Optional Hardware section of this document for the list of devices supported by drivers.)

A variety of other programming aids are supplied, such as template device drivers, declarations of Data Access Protocol (DAP) interfaces, and declarations of exception arguments.

The Toolkit is delivered with complete user documentation, including reference manuals for the VAXELN Pascal language and the VAXELN C runtime library.

▪ VAX-11 RSX

VAX-11 RSX is an emulator for RSX-11 operating system family and executes on all VMS and MicroVMS systems. It runs in compatibility mode on processors that support a PDP-11 instruction set subset in hardware or microcode and it also runs on certain processors without this support by providing its own software emulation of the same PDP-11 instruction set subset. It provides special capabilities that enable PDP-11 users to develop programs for execution in any of the following environments:

- VAX/VMS compatibility mode
- MicroVAX II/MicroVMS (software-emulated compatibility mode)
- VAXstation II/MicroVMS (software-emulated compatibility mode)
- RSX-11M-PLUS
- RSX-11M
- RSX-11S
- Micro/RSX
- P/OS

VAX-11 RSX also allows for the migration of many existing RSX-11 applications to VAX/VMS and MicroVMS.

Program Development Capabilities

The program development facilities provided by VAX-11 RSX consist of

- The PDP-11 Instruction Set Emulator (CEM$EMULATOR) which emulates the PDP-11 machine instruction set and allows RSX-11 tasks to run on MicroVAX II and VAXstation II processors that do not contain the compatibility mode hardware.
- The MCR command line interpreter (CLI). This CLI emulates the RSX-11 MCR CLI so that you can interact with a familiar user interface. MCR also provides access to many of the native VAX/VMS and MicroVMS program development facilities.
- The RSX-11 Application Migration Executive (AME) that emulates the RSX-11 Executive services.
- The Indirect Command File Processor (ICM) that allows RSX-11 indirect command files to be executed on VAX/VMS and MicroVMS.
- The DCL command back translator (BACKTRANS) that allows RSX-11 utilities to be invoked through the use of the DCL command interface.
- A subset of the RSX-11 program development utilities and libraries.
- A subset of the RMS-11 Version 2.0 program development utilities and libraries.

The following RSX-11 program development utilities are available to users of VAX-11 RSX:

- BRU – Backup and Restore Utility
- CRF – Cross Reference Processor
- DMP – File Dump Utility
- DSC – Disk Save and Compress Utility Program
- EDI – Line Text Editor
- FLX – File Transfer Utility Program
- LBR – Librarian Utility Program
- MAC – PDP-11 MACRO-11 Assembler
- PAT – Object Module Patch Utility
- PIP – Peripheral Interchange Program
- SLP – Source Language Input Program
- TKB – Task Builder
- ZAP – Task/File Patch Program

The following RSX-11 program development libraries and components are available to VAX-11 RSX users:

- FCSRES.STB – File Control Services symbol table
- FCSRES.TSK – File Control Services resident library
- ODT.OBJ – On-Line Debugging Tool object module
- QIOSYM.MSG – Standard RSX-11 QIO error messages
- RSXMAC.SML – Standard RSX-11 macros
- SYSLIB.OLB – System object library (non-ANSI version)
- VMLIB.OLB – Virtual memory subroutine library

The following RMS-11 program development utilities are available to VAX-11 RSX users:

- BCK – File Backup Utility
- CNV – File Conversion Utility
- DEF – File Definition Utility
- DES – File Design Utility
- DSP – File Display Utility
- IFL – Index File Load Utility
- RST – File Restore Utility

The following RMS-11 program development libraries and components are available to VAX-11 RSX users:

- RMSLIB.OLB – RMS-11 object library
- RMSMAC.MLB – RMS-11 macro library
- RMS11.ODL – Prototype disk-based overlay descriptor
- RMS11S.ODL – Minimum-size partial-function overlay descriptor
- RMS11X.ODL – Minimum-size full-function overlay descriptor
- RMS12S.ODL – Medium-size partial-function overlay descriptor
- RMS12X.ODL – Medium-size full-function overlay descriptor
- DAP11X.ODL – Full-function including remote support overlay descriptor
- RSMDES.IDX – Help file for the RMS DES utility

The following utility for file transfer to and from Micro/RSX systems is available to VAX-11 RSX users:

- MFT – Micro/RSX File Transfer Utility

General Access

When using VAX-11 RSX, you can gain access to the system through the normal VMS LOGINOUT procedure. You can request MCR as your command line interpreter (CLI) or have it specified as the default CLI in your User Authorization File. You can also use the VMS CLI, DCL. Under DCL, however, not all of the RSX and RMS program development utilities are directly available. Some of the utilities are available through DCL commands (LIBRARY/RSX11, for example,) but for some other utilities, you must explicitly request the utility to execute by typing RUN SYS$SYSTEM:utility-name or MCR utility-name.

VAX-11 RSX indirect command files may be executed from either MCR or DCL but must contain only indirect directives and MCR commands.

On VMS and MicroVMS it is only possible to switch from one CLI to another by logging out of the current CLI and then logging in again using the new CLI or by using the DCL or MCR SPAWN command. This differs from RSX-11M and RSX-11M-PLUS.

Disk and Tape Volumes

In addition to a native disk file structure, VMS and MicroVMS also provide a disk file structure (called Files-11 Structure Level 1) that is compatible with RSX-11. This provides for easy cross migration of code and data. Both file structures are available to programs running in either compatibility or native mode.

VAX-11 RSX supports general access to magnetic tape volumes. Tapes created on an RSX-11 system by BRU, DSC, FLX, PIP, and RMS BCK can also all be read on VAX-11 RSX by corresponding utilities. Similarly, it is possible to create tapes on VAX-11 RSX to be read on an RSX-11 system.

Intersystem Facilities

VAX-11 RSX includes support for the Micro/RSX Data Terminal Emulator (DTE) and File Transfer Program (MFT). Two-way file transfer capabilities to and from Micro/RSX systems are provided by the MFT utility supplied with VAX-11 RSX that executes on the VMS or MicroVMS system and the DTE utility that executes on the Micro/RSX system. Files of any type or size can be transferred from one system to the other in this manner. However, the file transfer process can only be initiated from the Micro/RSX system. The Micro/RSX user connects to the VMS or MicroVMS host system via a serial terminal line using the Micro/RSX Data Terminal Emulator (DTE) utility. The Micro/RSX user can then log into the host system as though the user's terminal were directly connected to the host.

DECnet is not available to RSX programs executing under VAX-11 RSX with one exception: applications written to use RMS-11 Version 2.0 will have full access to DECnet.

Compatibility

This product is an emulator of the RSX-11 family of operating systems. Specifically, this product is designed to emulate:

- RSX-11M-PLUS Version 3.0
- Micro/RSX Version 3.0
- RSX-11M and RSX-11S Version 4.2

As of Version 2.0, VAX-11 RSX now supports:

- Memory Resident Overlays
- Cluster Libraries
- FCSRES
- Search lists consisting of only devices and rooted directories
- VAXcluster SYSCOMMON (common system disk)

MCR Compatibility

The following RSX-11 MCR commands are supported:

ALLOCATE	ASN	BYE	CANCEL
DEALLOCATE	DEBUG	DMOUNT	EDT
HELP	INIT	MOUNT	RESUME
RUN	TIME	UFD	

The following VMS DCL commands are also available from the MCR CLI:

APPEND	ATTACH
CONTINUE	COPY
CREATE	CREATE/DIRECTORY
CREATE/NAME_TABLE	DEASSIGN
DEFINE	DELETE
DEPOSIT	DIFFERENCES
DIRECTORY	DUMP
EXAMINE	LOGOUT
MAIL	MERGE
PRINT	PURGE
RENAME	RUNOFF
SEARCH	SET
SHOW	SORT
SPAWN	STOP
SUBMIT	TYPE

The installation procedure provides the option to install an MCR help library that contains help text on both the RSX MCR and DCL commands, part of the MCR CLI.

Indirect Command File Compatibility

All indirect command file directives and functions are supported to some extent except .FORM, .WAIT, and .XQT. Most RSX-11 indirect command files can be executed successfully on VAX-11 RSX.

The following system generations and network generations are specifically supported:

- RSX-11M-PLUS Version 3.0
- RSX-11M Version 4.2
- RSX-11S Version 4.2
- DECnet-11M-PLUS Version 3.0
- DECnet-11M Version 4.2
- DECnet-11S, Version 4.2

Note

MicroVAX II and VAXstation II are NOT recommended for RSX-11 system generations or DECnet network generations due to the performance characteristics of the PDP-11 instruction-set emulator on these processors.

General Areas of Incompatibility

Every effort has been made to make the functions VAX-11 RSX supports as compatible as possible with the RSX-11 environment. However, certain areas of incompatibility do exist in this product and may continue to exist in future versions. The few areas of incompatibility mentioned in the various sections are not guaranteed to be all inclusive.

Other areas where incompatibilities exist include:

- No support for supervisor mode libraries
- No support for I-and-D space separation
- There are several differences between RMS-11 and VAX RMS

Compatibility with Other Derivatives of RSX-11

P/OS support is limited to the P/OS directives which are identical to RSX-11M-PLUS directives and are listed in the table above.

No compatibility is expressed or otherwise implied with any other versions of the RSX-11 family of operating systems or related operating systems, except where specifically noted.

Optional Software

The following optional software products require VAX-11 RSX as a prerequisite for being generated or run on VMS and MicroVMS systems.

ALL-IN-1 Office Menu V1.4 (Form Editor Application only)
CORAL 66/VAX to RSX Cross Compiler
DECnet-11M (network generation only)
DECnet-11M-PLUS (network generation only)
DECnet-11S (network generation only)
FORTRAN IV/VAX to RSX
MicroPower/Pascal-VMS
PDP-11 DATATRIEVE/VAX
PDP-11 FORTRAN-77 DEBUG/VAX to RSX
PDP-11 FORTRAN-77/VAX to RSX
PLXY-11/VAX
Professional Host Communications
Professional Host Tool Kit
Professional Host Tool Kit BASIC-PLUS-2
Professional Host Tool Kit COBOL-81
Professional Host Tool Kit DIBOL
Professional Host Tool Kit FORTRAN-77
Professional Host Tool Kit FORTRAN-77 DEBUG

Professional Host Tool Kit Pascal
Professional Real Time Interface Library
RSX-11M (system generation only)
RSX-11M-PLUS (system generation only)
RSX-11S (system generation only)
RTEM-11
VAX CORAL 66

▪ MicroPower/Pascal-VMS: Modular Executive and Microcomputer Software Development Toolset

MicroPower/Pascal-VMS is a VAX/VMS optional program development product. MicroPower/Pascal is a modular executive and software development package for PDP-11 (Q-bus) based microcomputer applications. It includes software components needed to create, build and debug/test concurrent real-time application software running stand-alone on a target runtime microcomputer system.

MicroPower/Pascal-VMS supports application software development using two distinct hardware environments:

- Host VAX/VMS development system
- Target PDP-11 (Q-bus only) runtime system for the microcomputer application

The application software is created and linked with the appropriate MicroPower/Pascal runtime software components on the VMS host development system. When the application software is ready for debugging/testing, it is transported to the target runtime system. The application software can then be executed on the target runtime system. If a serial line is connected between the host development system and the console port of the target runtime system, the execution of the application software in the target can be controlled and tracked from the host with the help of the debugging tools provided in MicroPower/Pascal-VMS.

The separation of the host development system from the target runtime system allows you to make use of a high-performance VAX/VMS host development system and test the application software in the target runtime environment. The MicroPower/Pascal-VMS Software Package includes the following components:

- MicroPower/Pascal-VMS Installation/Verification command procedures
- Host-Development System Software components
  - Extended Pascal compiler
  - Symbolic debugger
  - MicroPower/Pascal-VMS Utilities
  - MPBUILD Application Building Command Procedure
- MicroPower/Pascal target Runtime System Software
- Modular realtime executive
- Target runtime device handlers
- RT-11 compatible file system
- Timer services (clock process)
- Pascal Object Time System (OTS) for target
- MACRO-11 source libraries

An extended version of Pascal is provided as the system implementation language suitable for most user applications. The extensions in the language enable the user to write realtime applications entirely in Pascal; however, MACRO-11 can also be used as the implementation language for sections of code.

Several components of MicroPower/Pascal-VMS (such as the MicroPower/Pascal-VMS Utilities and the MicroPower/Pascal-VMS Pascal Compiler) run under VAX-11 RSX.


VAX
BASIC
VAX
COBOL
VAX
PASCAL
VAX
DBMS
VAX
RPG II
VAX
BLISS
VAX
CDD
VAX
Ada

Chapter 7 ▪ Language and Tool Integration in the VAX/VMS Software Development Environment

▪ Overview

There are many approaches to developing software products; this chapter gives you an overview of how many of our teams develop software at Digital. Specifically, it shows you how VAX/VMS languages, tools, and VMS services and program development utilities are integrated to create a consistent software development environment.

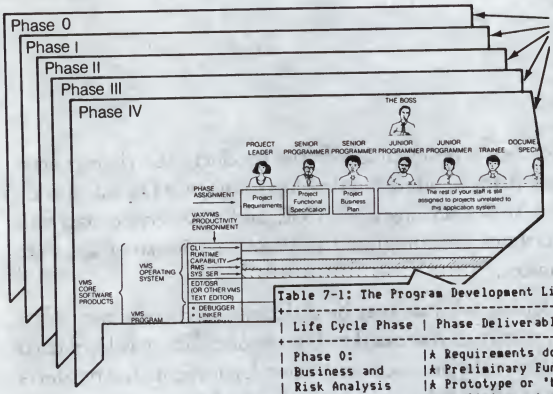
Your method of developing software may be different. Maybe you use only a few of the steps and techniques discussed in this chapter. You may have been looking for ways to enhance the productivity of your department. The products discussed in this chapter might help you achieve that goal.

If you're interested in using a highly productive environment of software development tools, then welcome to the VAX/VMS Software Development Environment.

Topics in this chapter include

- An outline of the product development life cycle used within Digital.
- An example of how we use VAX languages, tools, and VMS Services and Program development utilities in various stages of the software life cycle.

THIS CHAPTER:

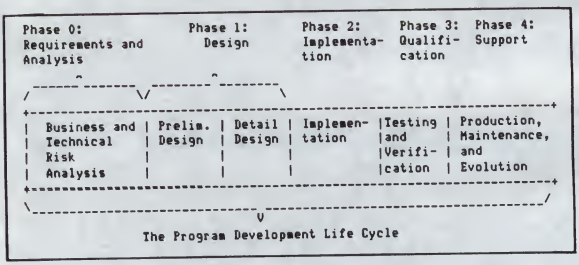


Provides you with a specific example of how a shop of seven developers creates a hypothetical software system using VAX/VMS products in the five phases of the program development life cycle.

Identifies deliverables generated in each phase of the life cycle and the products from the VAX/VMS Environment used to produce those outputs.

Table 7-1: The Program Development Life Cycle

Life Cycle Phase	Phase Deliverable
Phase 0: Business and Risk Analysis	▪ Requirements document ▪ Preliminary Functional specification ▪ Prototype or "breadboard" (sometimes) ▪ Preliminary testing strategy ▪ Technical analysis
Phase 1: Design	▪ Design Document ▪ Final functional specification ▪ Final Development Plan (Project schedule) ▪ Test specification ▪ "Early product" code
Phase 2: Implementation	▪ Code modules ▪ Debug modules ▪ Updated project documents ▪ Intermediate working versions of the system (Baselevels) ▪ User documents ▪ Unit tests ▪ Performance Testing ▪ System Testing ▪ Final Test Modules ▪ Field Test Kit ▪ Write a maintenance document
Phase 3: Qualification	▪ Customer Environment Test ▪ Final Product (Sw/Doc) Kit
Phase 4: Production, Maintenance & Evolution	▪ Archive Copies of Sources & Documents ▪ Fix bugs, when reported ▪ Enhance the product, if justified



Provides you with a detailed description of each phase in the program development life cycle, as we define it a Digital.

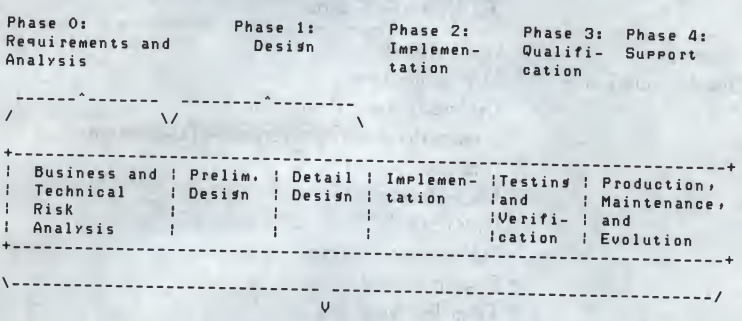
Figure 7-1 ▪ Overview of Chapter 7

Introduction

We are going to look at a specific example of how the VAX languages, VAX tools, VMS services, and program development utilities can be collectively applied to solving program development problems. First, let's review the program development life cycle and take a close look at each phase of that life cycle. We will review the output a programming department might produce in each of those phases, and the VAX languages, tools, and program development utilities that can be used.

The Program Development Life Cycle

As you can see below, we have chosen to define the program development life cycle in terms of five phases. Some of these are further divided into more specific areas. These may vary from company to company. The general concept, however, is valid in the design of any software system.



The Program Development Life Cycle

Table 7-1 shows us some of the typical output generated in each phase.

Table 7-1 ▪ The Program Development Life Cycle

Life Cycle Phase	Phase Deliverables
Phase 0: Business and Risk Analysis	* Requirements document * Preliminary Functional specification * Prototype or "breadboard" (sometimes) * Preliminary testing strategy * Technical analysis * Business Plan
Phase 1: Design	* Design Document * Final functional specification * Final Development Plan (Project schedule) * Test specification * "Early product" code
Phase 2: Implementation	* Code modules * Debug modules * Updated project documents * Intermediate working versions of the system (Baselevels) * User documents * Unit tests * Performance Testing * System Testing * Final Test Modules * Field Test Kit * Write a maintenance document
Phase 3: Qualification	* Customer Environment Test * Final Product (Sw/Doc) Kit
Phase 4: Production, Maintenance_& Evolution	* Volume reproduction of final product * Archive Copies of Sources _& Documents * Fix bugs, when reported * Enhance the product, if justified

These deliverables can be categorized as documents, programs, tests, and files. We use combinations of software tools to specifically manage deliverable in each category. For example, we can use the Language Sensitive Editor to call up templates for documents (as well as programming languages), DEC/CMS to track changes in documents, the VAXTPU editor to actually make changes, DSR to format chapters of each document, and DEC/MMS to actually control the construction of one or more documents.

A brief description of each Engineering Phase is given below. A detailed explanation of how VAX/VMS tools, languages, and core development utilities can be used during a phase to produce the required output is found in the second section of this chapter, beginning under the subhead, "Here's a Specific Example".

- PHASE 0: BUSINESS AND RISK ANALYSIS

  Project definition is the first phase in the product development life cycle. During this phase a team of technical and product management people defines business opportunities, product objectives, and technical options. The cost versus benefit of the project is analyzed. At the completion of this phase, the project team has defined project goals and has written a requirements documents, a business plan, and perhaps a preliminary functional specification document. The technical team also makes sure that they will be able to confirm correct operation of the potential product.

  During the analysis portion of this phase, the project team is determining technical approaches needed to build the intended product. Often, possible solutions to difficult technical problems are breadboarded or prototyped to make sure the implementation risks are well understood. Prototyping is especially important in understanding the human interface to the software and its ability to be used. At the end of this stage of development, the system is generally defined, and a business decision is made on whether to attempt an actual project start.

- PHASE 1: DESIGN

  During this phase, the project team determines precisely what has to be built and how it will be accomplished. The first step in this phase is writing the final Functional Specification and the Development Plan (schedule), and Documentation plan. When project specifications are complete, design can then take place and the software product takes on full-system definition (Note, this chapter uses "design" in the same context that many organizations use "analysis").

  Top-level designs for all forms, data structures, program modules, file formats, and human interfaces are made based on the items listed in the functional specification. Once completed, the design gives the project technical definition.

A Design Specification and a Test Plan, generated during this phase, serve as a basis for acceptance of the design. A design document makes it possible to keep the design specifications in one location, accessible by all programmers. As the project design evolves, so does the design document.

- PHASE 2: IMPLEMENTATION

  The implementation phase of the program development life cycle is most often associated with building source code modules, then compiling, linking, and executing the resulting images. Often, the system is implemented in a series of stages or baselevels in which each baselevel adds more and more of the required functionality. However, because program development is an evolutionary process, many other steps also occur concurrently.

  During the implementation phase, user documentation and the test generation also run at full speed. The tests defined in the requirements document must be created and run to complete ensure a correctly operating implementation. Unforeseen problems in implementing design requirements might mean that specifications and designs require rework. If this is the case, then the effects of changes in requirements or designs must be reflected accurately in both the programs being written and the user documentation set. At successful completion of this phase, the project should have a software system that works.

  The project team must analyze the structure and performance of the software in this Phase, in addition to passing functional tests without errors. During this phase, design, code, and documentation reviews are held frequently. Other groups can be given copies of the software to determine how well the program works under controlled conditions. Performance analysis ensures the system will meet certain customer-environment requirements.

- PHASE 3: QUALIFICATION

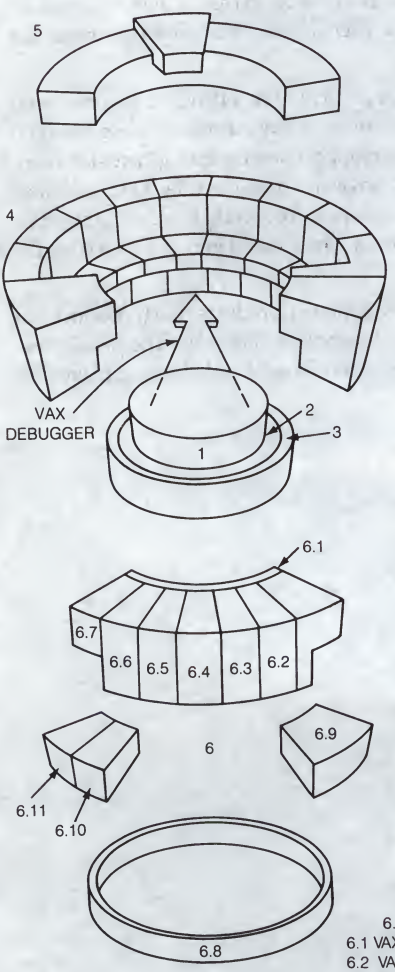
  During this phase, the software is in use in selected customer environments. The technical team stays in close contact with external test sites, making sure any needed corrections are reflected in the version of the software and/or documentation to be shipped to our general customer base. In later stages of this test period, the sources and documentation are frozen, and final copies of the books and distribution media are prepared.

- PHASE 4: VOLUME PRODUCTION, MAINTENANCE AND EVOLUTION

  In this phase, master copies of the documentation and the software are handed to a high-volume production group for replication and distribution to Digital Software Specialists (support groups) as well as customers with support contacts in effect. Copies of the master kit are archived, in case of physical disasters. A post-mortem review is held on the overall project development cycle just completed.

  After the product has been shipped, a process of maintenance and evolution begins. Should there be errors in the software or documentation, bug fixes and updates will be made. Enhancements to support new operating system or hardware releases may be planned. Suggestions arrive from customers using the new product. In fact, because a software system is frequently evolving, this phase becomes an information-gathering activity which could initiate Phase 0 for the next version of the software.

Figure 7-2 lists VAX/VMS program development products. Each product can play an important part in the program development life cycle. The specific features and benefits of each of these products have already been explained in earlier chapters of this handbook.



VMS CORE SERVICES

1. The VMS Operating System

2. VMS Services
   - The Digital Command Language (DCL)
   - The VMS Runtime Library (RTL)
   - VMS system services
   - The VAX Record Management Services (RMS)

3. The VMS Program Development Utilities
   - The VAX/VMS Symbolic Debugger
   - Sort/Merge
   - The VAX Linker
   - The VAX Librarian
   - VAXTPU text processing utility
   - DSR text-formatting utility
   - Mail

VAX OPTIONAL PROGRAM DEVELOPMENT PRODUCTS

4. VAX Languages
   - VAX Ada
   - VAX Apl
   - VAX BASIC
   - VAX BLISS
   - VAX C
   - VAX COBOL
   - VAX CORAL
   - VAX DIBOL
   - VAX DSM
   - VAX FORTRAN
   - VAX LISP
   - VAX PASCAL
   - VAX PL/I
   - VAX RPG II
   - Other VAX Languages

5. VAX Software Productivity Tools
   - VAX DEC/Code Management System (CMS)
   - VAX DEC/Module Management System (MMS)
   - VAX Language-Sensitive Editor
   - VAX Performance and Coverage Analyzer (PCA)
   - VAX DEC/Shell
   - VAX DEC/Test Manager
   - VAX Graphical Kernel System (GKS)

6. Related Program Development Products

6.1 VAX CDD	6.7 VAX FMS
6.2 VAX DBMS	6.8 DECnet Communications Products
6.3 VAX Datatrieve	6.9 VAXELN
6.4 VAX Rdb/VMS (ELN)	6.10 VAX-11 RSX
6.5 VAX ACMS	6.11 MicroPower/Pascal-VMS
6.6 VAX TDMS	

Figure 7-2 ▪ The VMS operating system, VAX Languages, VAX Tools, Program Development Utilities, and Related VAX Software Products

Table 7-2 ▪ VAX/VMS Products Used In The Software Development Life Cycle

Life Cycle Phase	Deliverables Generated During The Phase	VAX/VMS Service or Product Used
Phase 0 Business and Risk Analysis	* Requirements document * Preliminary Functional Specification * Business Plan * Prototype or "breadboard" (sometimes) * Preliminary testing strategy * Technical analysis	* text editor(s) and DSR text-formatting utility * Mail utility * Language Sensitive Editor (document templates) * DEC/MMS * DEC Test Manager * DEC/CMS
Phase 1 Design	* Design Document * Final Functional Specification * Final Development Plan (schedule) * Test specification * "Early Product" code	* System Services * Common Run Time Library * Record Management Services * VAXTPU Editor and DSR text-formatting utility * Mail utility * Language Sensitive Editor (document templates) * Common Data Dictionary (CDD) * FMS (Forms Management) or VAX TDMS * DEC/CMS * DEC/MMS * DEC/Shell or DCL * Language Compilers

(continued on next page)

Table 7-2 ▪ VAX/VMS Products Used In The Product Development Life Cycle (Cont.)

Life Cycle Phase	Deliverables Generated During The Phase	VAX/VMS Service or Product Used
Phase 2 Implementation	* Code modules * Debug modules * Final Test Modules * Intermediate working version of the System (Baselevels) * Unit Tests * User documents * Updated project documents * Performance Tests * System Tests * Field test kit * Maintenance Document	* System Services * Common Run Time Library * Common Data Dictionary (CDD) * FMS (Forms Management) or VAX TDMS * Record Management Services * VAXTPU Editor and DSR text-formatting utility * Mail utility * Debugger * Linker * Language Sensitive Editor * DEC/Test Manager * Performance and Coverage Analyzer (PCA) * DEC/Shell or DCL * DEC/CMS * DEC/MMS * Language Compilers
Phase 3 Qualification	* Use in a customer environment * Final Product Kit (SW/Doc)	* Debugger * text editors * DEC/Test Manager * Performance and Coverage Analyzer * Language Compilers * Language Sensitive Editor
Phase 4 Production, Maintenance, and Evolution	* Archive a copy of the master kit * Fix bugs, when reported * Enhance the product, if justified	* All services and products listed above

▪ Here's a Specific Example

The rest of this chapter is an example of how you might use VAX/VMS tools and languages in your department to produce software. We will follow the development of a hypothetical software system through each phase of the program development life cycle (defined earlier in this chapter).

Keep in mind the purpose of this example. It gives you the opportunity to see how we use VAX/VMS products to develop software at Digital. It has also been provided to emphasizes the unique ability of our products to help you maintain and control the base of knowledge that must be managed if you are to program productively.

Your Department

Figure 7-3 introduces you to your department — six programmers and a documentation specialist; all are working on a VAX processor running the VMS operating system (services and program development utilities included), multiple VAX languages, and productivity tools installed. Note that these software products are organized vertically along the left side of Figure 7-3.

Team member 1 is the Project Leader for the software system your department is about to begin work on. Members 2 and 3 are senior programmers, while 4, 5, and 6 are junior programmers. Team member 7 is the writer/documentation specialist.

Member 3 is a FORTRAN programmer and Member 4 has prior experience with COBOL. The rest of your department uses VAX C. It may, therefore, be necessary to write the software system in several languages if programmers 3 and 4 are to be as productive as possible. The VAX Common Language Architecture, as we will see later, makes it possible for a single application to be written using more than one VAX language.

Once again, in Figure 7-3, at the top of the column titled, VAX/VMS Program Development Environment, you will note there are a number of VMS services running across the entire figure, below all the programmers. These services — DCL, RMS, RTL, and VMS system services — are provided by the VMS operating system and are always present on the system. (For more on these services, see Chapter 4 of this handbook.)

The first VMS service (a horizontal bar directly below the assignment boxes) represents the command language interpreter used by all programmers. Programmer 1, because she comes from a Unix environment, is using the VAX DEC/Shell command language interpreter in conjunction with DCL.

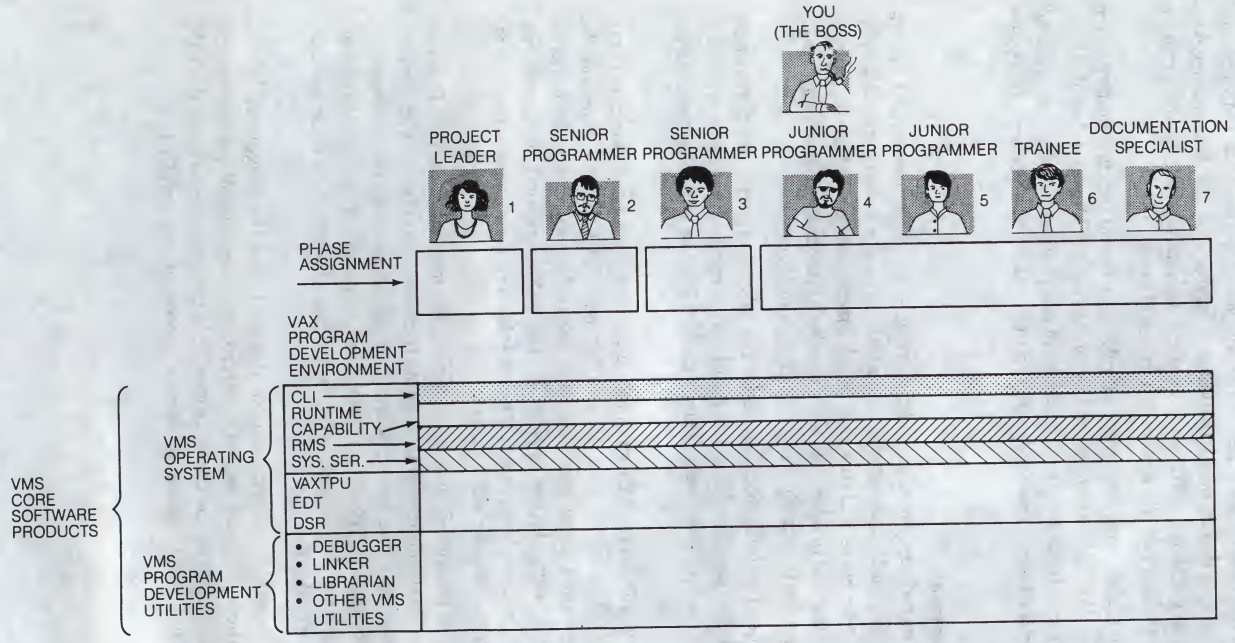


Figure 7-3 ▪ A Model Program Development Department

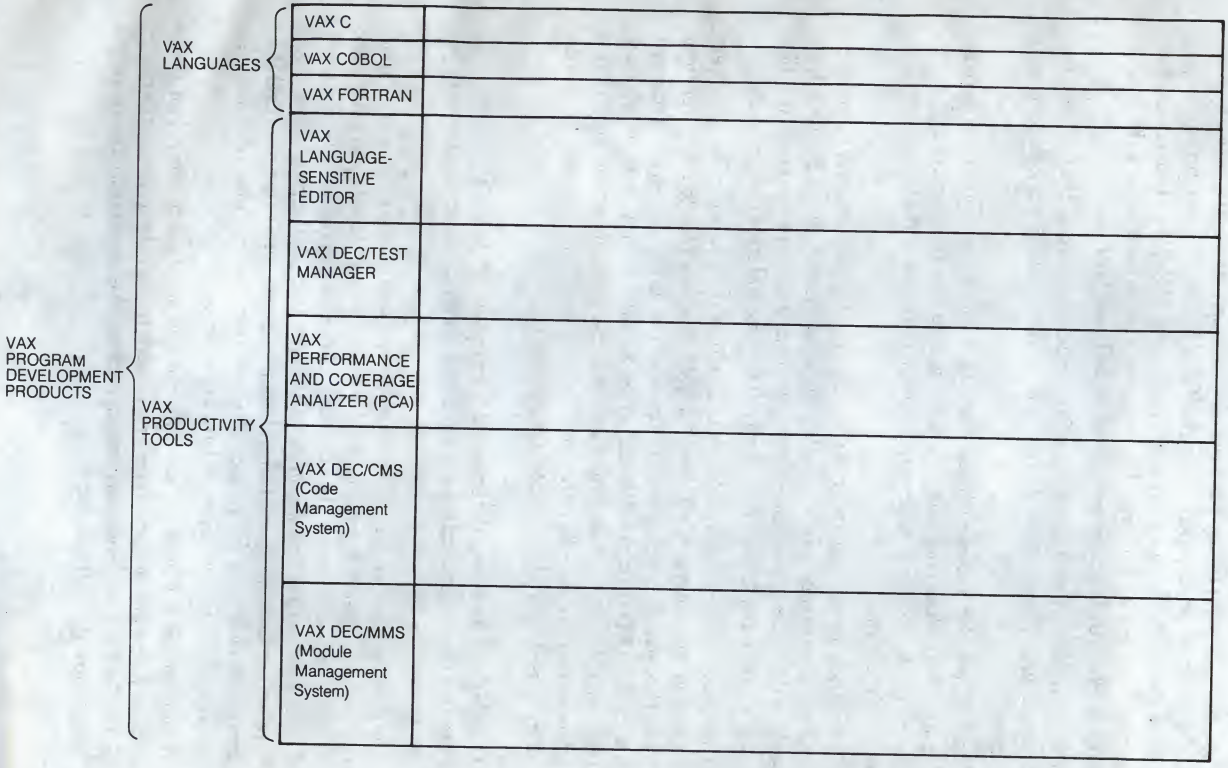


Figure 7-3 (Cont.) ▪ A Model Program Development Department

Getting Started

Figures 7-4 through Figure 7-8 show us your programming staff in each phase of the program development life cycle. These figures illustrate project assignments, the work flow of each assignment, and VAX/VMS products used to accomplish those assignments. For example, Figure 7-4 illustrates the flow of work in Phase 0 of the life cycle, Business and Risk Analysis.

The general layout of these figures is important to understand before you continue reading. They serve as the basis for the discussion of work done in each phase.

You will note that below each team member's picture in Figures 7-4 through Figure 7-8 is a block containing his/her assignment for that particular phase. In Figure 7-4, for example, under programmer 1's picture is the assignment: project requirements.

Directly below that assignment block is the first step in a sequence of steps she must perform to successfully complete the assignment. Again, in Figure 7-4, we see that the first step in writing the requirements document is to create text files (DSR source-input ".RNO" files) in which to write and store the document.

The names of the primary VAX/VMS products used to complete a step are located in the column titled, the VAX/VMS Program Development Environment. (Located at the left side of the figure.) Thus, in Figure 7-4, programmer 1 has used VAXTPU or the VAX Language-Sensitive Editor (with appropriate template files), to create and store text in a DSR input file. (Note the VAXTPU editor is used to create the DSR input ".RNO" files and the DSR text-formatting utility is used to produce ".MEM" output files.)

Defining and Analyzing the Software System with VAX/VMS

Now that you understand the format of these figures, we can continue with our example.

The definition and analysis of a new software system is one of the most important parts of a software system's development. During phase 0, you must make many important decisions that will affect subsequent phases of the life cycle. Therefore, it is of prime importance that input from all the programmers and support personnel be quantified and organized into output that is accessible to everyone involved in the project. Doing it right the first time minimizes the amount of fine tuning necessary later in the life cycle.

As you can see in Figure 7-4, you and your staff are about to begin. Not all the staff members are involved in the new project. (Notice that team members 4, 5, 6, and 7 are still working on a previous project.) Assignments for this phase are listed below the staff member.

Defining and Analyzing the Software System with VAX/VMS

Now that you understand the format of these figures, we can continue with our example.

The definition and analysis of a new software system is one of the most important parts of a software system's development. During phase 0, you must make many important decisions that will affect subsequent phases of the life cycle. Therefore, it is of prime importance that input from all the programmers and support personnel be quantified and organized into output that is accessible to everyone involved in the project. Doing it right the first time minimizes the amount of fine tuning necessary later in the life cycle.

As you can see in Figure 7-4, you and your staff are about to begin. Not all the staff members are involved in the new project. (Notice that team members 4, 5, 6, and 7 are still working on a previous project.) Assignments for this phase are listed below the staff member.

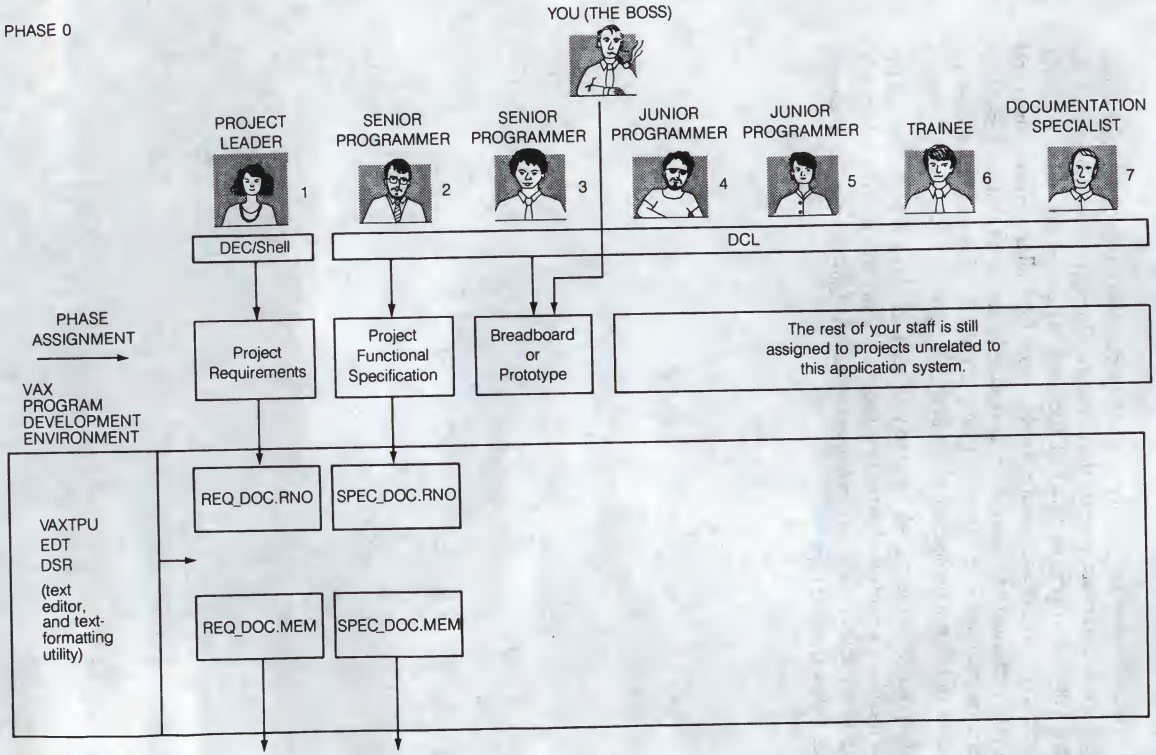


Figure 7-4 ▪ Phase 0 — Defining and Analyzing the Software System

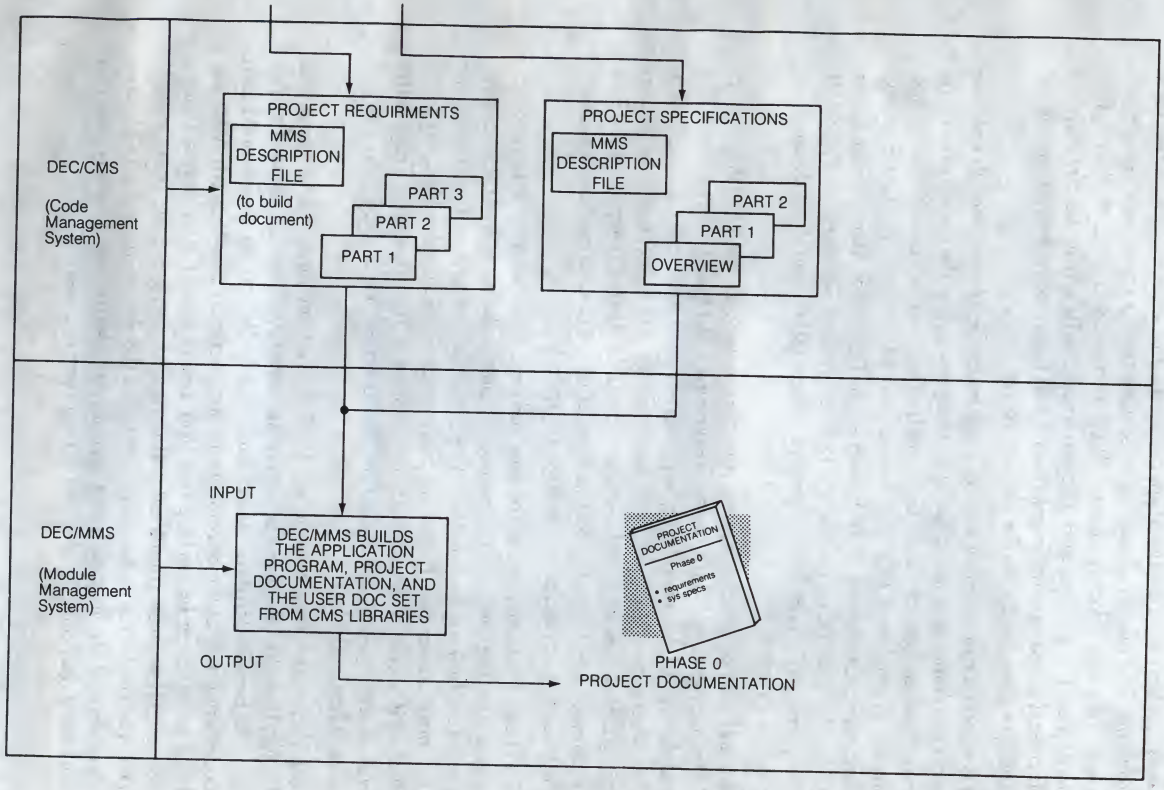


Figure 7-4 (Cont.) ▪ Phase 0 — Defining and Analyzing the Software System

You and your three senior programmers will generate the project requirements, preliminary functional specification, and analyze difficult technical problems. The output of each of these assignments represents a single section in the project documentation. Programmer 1 will generate the requirements document, you and Programmer 2 will write the preliminary functional specification. Programmer 3 will begin to work on the technical problems associated with the overall project.

It is important to note that each of these assignments, even though only one person is ultimately responsible for its completion, is really a team project. All team members generate input that must be incorporated into each other's documents. A number of VAX/VMS tools are available to help you better manage these paperwork aspects of the project — the DEC/Code Management System (CMS), the VAXTPU text editor, the DSR text-formatting utility, the VMS MAIL utility, and DEC/Module Management System (MMS) and the VAX Language Sensitive Editor.

You and each programmer create your documents with the VAXTPU editor or Language Sensitive Editor (using document-related templates specific to DSR and the type of document(s) to be produced) and DSR text-formatting facility. Each document then becomes a require file for a DSR document. As they are created, these files are stored in one or more CMS libraries. Then, once you have determined how to put the document together, DEC/MMS can build or rebuild the versions you require with a single command.

CMS helps track the reasons, the time and date, and the initiator for modifications to the documentation. A number of intermediate stages of the document can be established. By consulting these, each person involved in creating the document can see exactly what the most current version of the document looks like, or, if necessary, reconstruct past versions of the document. This control enables those you choose to see how the document has evolved; who changed what, when, and why.

The Requirements document, the Preliminary Functional Specification, and Project Plan document can be part of the same CMS documentation library or each can be stored in its own CMS library. DEC/MMS can be used to build each document (or all of the documents, in one operation, if so desired). MMS can build a document by fetching files from one or more CMS libraries. Once a document has been completed and approved, you can issue commands to DEC/CMS to "freeze" a particular version of the document under a CMS-class name you specify. You also have the ability to make the now-frozen version of a document read-only, i.e. unalterable by project members.

One of the things often done in this phase is rapid prototyping of target system capabilities. The power of the Language Sensitive Editor and the various VAX language implementations — VAX Ada or VAX BASIC, for example, can be beneficial in this phase. The symbolic debugger works with the VAX languages to enable you to see actual source code in a window on your terminal–while you are debugging code running in another window. VAX DATATRIEVE can help with report generation during design. VAX COBOL has powerful screen-handling and report-writing capabilities. The Shell and the Digital Command Language (DCL) have powerful utility programs which are also very helpful in prototyping some types of applications.

Data structures, file formats, and forms are frequently prototyped in this phase. Many projects use the Common Data Dictionary to hold the definitions of major data structures. The dictionary saves this information in a language-independent form, and a valid declaration can be extracted by many of the VAX languages. This allows you to define a record structure, and access data using it, from programs written in BASIC, COBOL, DATATRIEVE, FORTRAN, PASCAL, or other languages–yet only a single declaration exists (in the dictionary). When building a system, DEC/MMS is capable of checking declarations stored in the Common Data Dictionary (CDD), and recompiling program modules which depend on records whose structure has been altered. Similarly, MMS can check forms libraries to make sure that programs which depend on forms are recompiled, if need be.

Designing the Software System with VAX/VMS

After the requirements for a software system are defined, you next start the actual design of the system. As you will note in Figure 7-5, more members of your staff have become involved in the process. During this phase, further work may be done to clarify the specification and the project plan, in addition to writing a design document for the project. Programmer 4 writes test specifications, and the documentation specialist starts on the user-documentation plan. All participate in design reviews.

With the VAX Language Sensitive Editor, you can create and then use templates for documentation formats and specifications. This eliminates the need to retype boiler-plated sections of your document and promotes consistency throughout your documentation — particularly when multiple team members are involved.

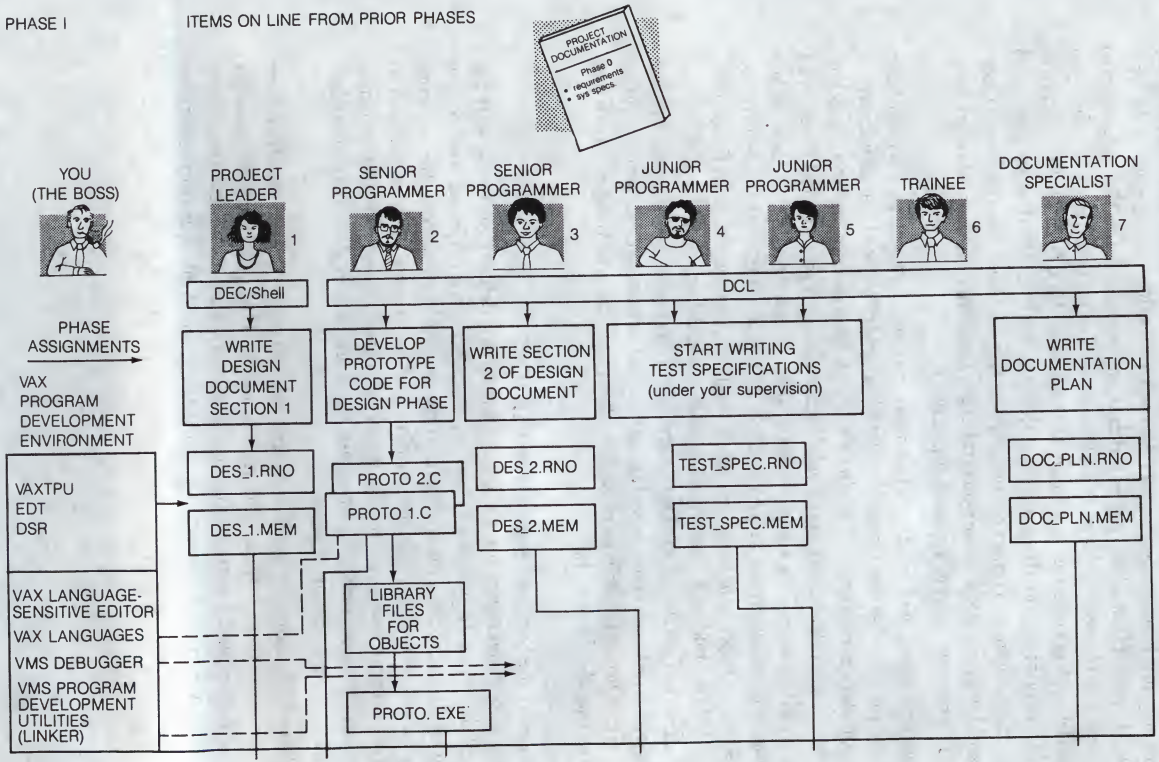


Figure 7-5 ▪ Phase 1 — Designing the Software System with VAX/VMS

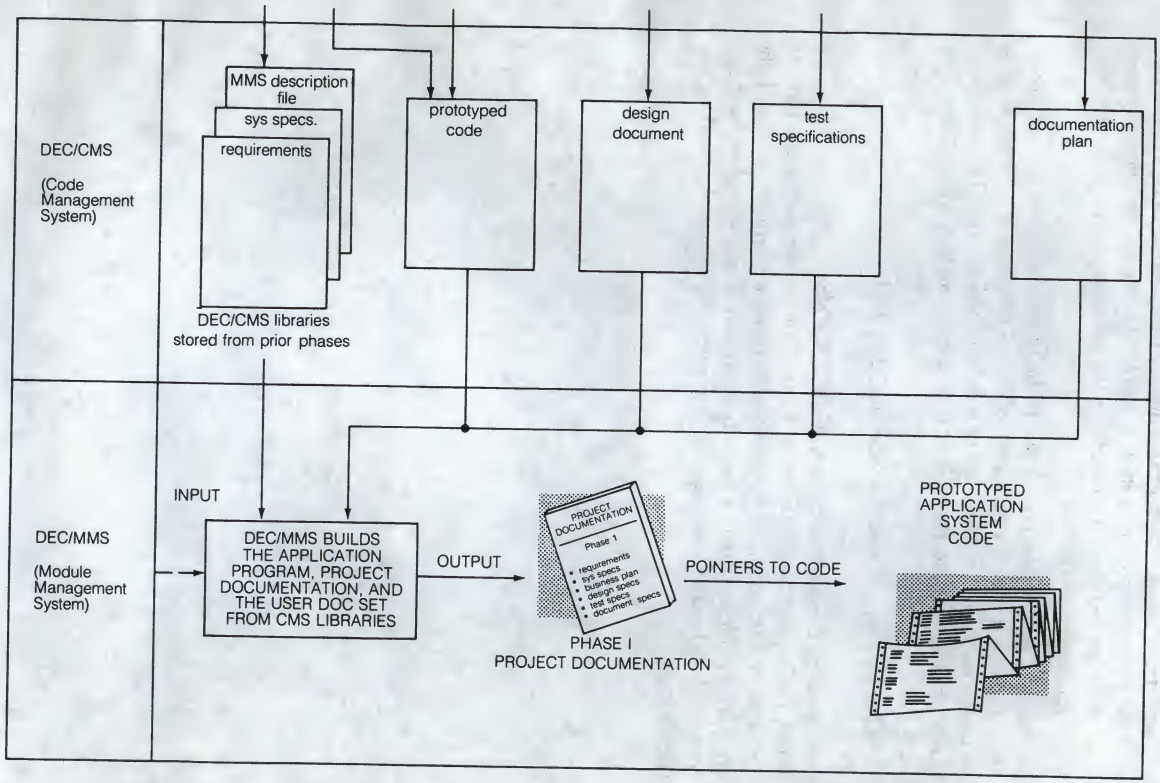


Figure 7-5 (Cont.) ▪ Phase 1 — Designing the Software System with VAX/VMS

Above the pictures of your programmers in Figure 7-5 (not found in Figure 7-4) are "Items Online From Prior Phases". For example, a representation of Phase 1 project documentation is there. This means the time-stamped documentation you developed in Phase 1 is online, and accessible to everyone in your department.

Any project documentation developed in this phase (the design and documentation plans) or necessary corrections to Phase 1 documentation (after all, this can be an evolutionary process) will be incorporated into Phase 1 project documentation and become Phase 2 project documentation when Phase 2 is finished.

(See Chapter 1 for a general overview of all these tools.)

Implementing the Software System With VAX/VMS

During the implementation phase (see Figure 7-6), you are making coordinated use of many VMS tools. All the personnel in your department are now fully involved in implementation of the software system. The design phase has been completed, and during this time, code is written to build a product to the given specifications.


VAX
PASCAL
VAX
ADA
VAX

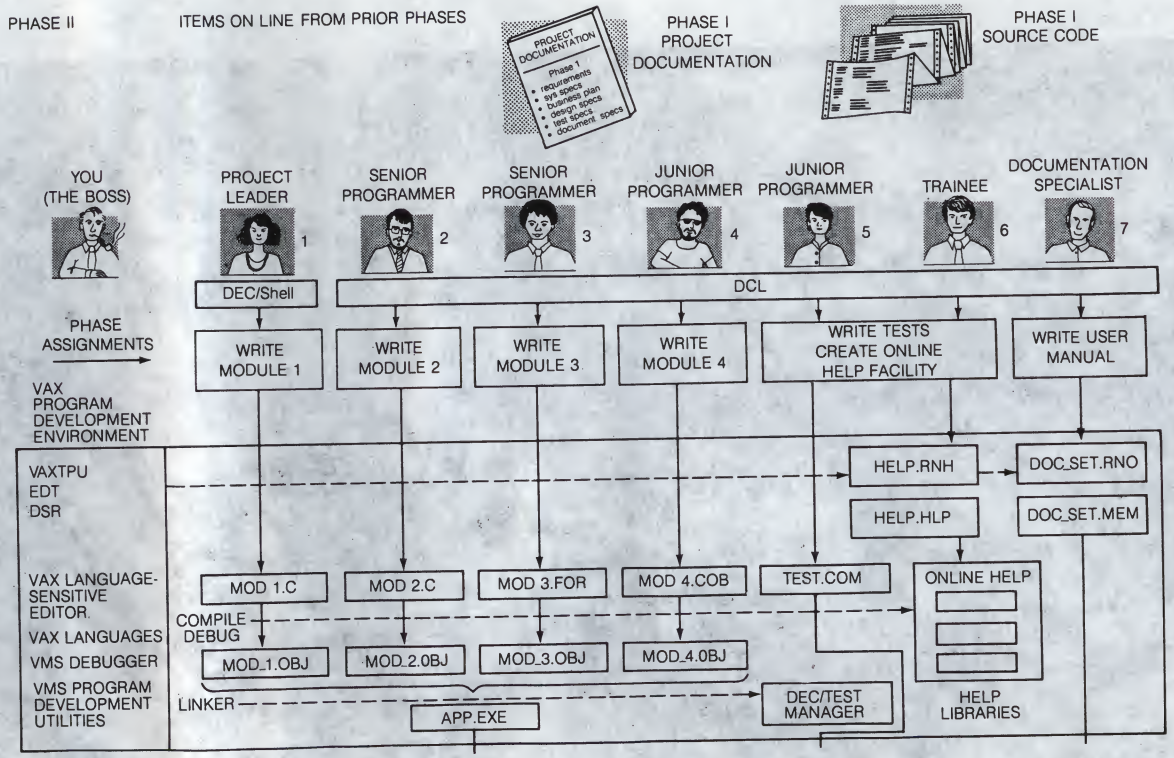


Figure 7-6 ■ Phase 2 — Implementing the Software System With VAX/VMS

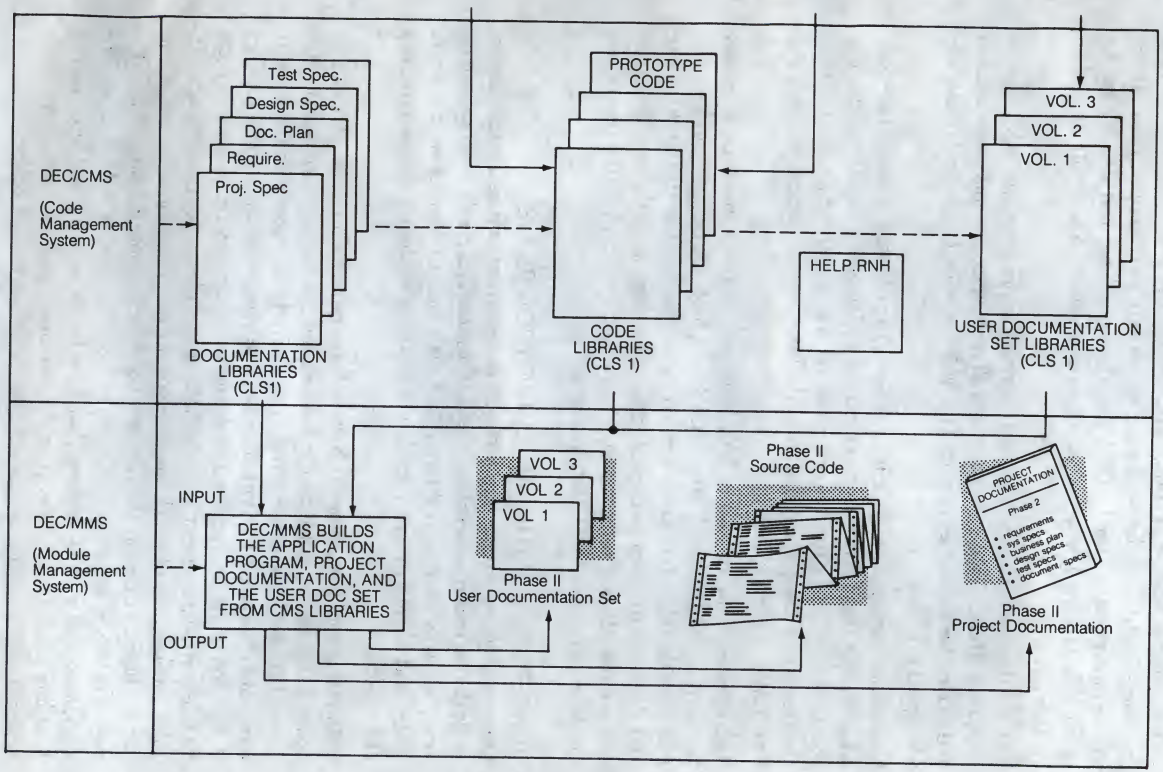


Figure 7-6 (Cont.) ▪ *Phase 2 — Implementing the Software System With VAX/VMS*

Assignments for this phase are as follows — programmer 1, 2, 3, and 4 each have to write a module of code, as outlined in the project document. Programmers 5 and 6, under your supervision, will start to write tests (These will be incorporated into the DEC/Test Manager.) They will also develop an online help facility. The writing of user documentation is in full swing during this phase.

Some of the most valuable tools used during this phase are the VAX language compilers. These high-quality compilers all meet or exceed industry standards and all have been designed as part of the VAX Common Language Architecture (discussed in Chapter 1).

Often implementation is done in several stages or baselevels. Each base level adds more of the specified features you are building into the software system. Sometimes, several programmers work on the same module. Source code changes can be easily controlled using the Reserve and Merge features of VAX DEC/CMS.

The integrated use of various VAX language compilers, the VAX Language Sensitive Editor, and the VAX Symbolic Debugger makes the edit-compile-link-debug process much more efficient. The VAX Performance and Coverage Analyzer finds performance bottlenecks in the emerging code and ensures sufficient test coverage. Programmers can perform test procedures quickly using the VAX DEC/Test Manager.

- VMS SERVICES IN THE IMPLEMENTATION PHASE

  Operating system services and program development utilities are used extensively during the implementation phase. These features are the VMS system services, the VAX Runtime Library (RTL), VAX Record Management Services (RMS), and the VAX program development utilities (including the powerful VAX Debugger).

  VMS offers your programmers System Services and Run Time Library procedures that can be called by the software module they are writing. The Run Time Library has procedures which include screen management, terminal-independent I/O, virtual memory management, and parsing routines, in addition to the language and mathematics support procedures one expects. VAX Record Management Services (RMS) makes file access consistent and efficient, independent of which language you are programming in. Access to files can be limited through the use of VMS Access Control Lists (ACLs). The powerful VAX SORT utility allows you to order files quickly and efficiently. This combination of System Services, Run Time Library utility procedures, and file handling procedures allows your programmers (and your design and code reviews) to concentrate on routines that are unique to their project.

Programmers sometimes use the Digital Command Language (DCL) to write tests, run tests, and prototype simple concepts. If a programmer is accustomed to a UNIX® -style environment, then VAX DEC/Shell provides them with a Bourne Shell with which to interface to VMS, and to many UNIX-style utilities (including pipes). The VAX DEC/Shell can be used as a script language for prototyping.

- USING THE VAX LANGUAGE SENSITIVE EDITOR DURING IMPLEMENTATION

The VAX Language-Sensitive Editor is a high-performance, screen-oriented editor with multiple windows and buffers for program development.

Highlights of the VAX Language Sensitive Editor are listed below.

- The VAX Language Sensitive Editor lets you code, edit, compile, review, and correct compilation errors within a single editing session. In review mode, compilation errors are displayed in one window of the editor with the corresponding source code presented in the second window.
- The VAX Language Sensitive Editor gives you DCL-like line-mode commands and a keypad layout that makes it easy for users familiar with VAXTPU to quickly become familiar with the VAX Language Sensitive Editor.
- The Language Sensitive Editor is fully integrated with VAX languages that have been designed to the common language architecture. The editor has a set of templates for the major constructs in each language. For example, you can insert an IF statement in your favorite language into a source file by simply typing IF and then a <CTRL/E> (for expand). A complete IF-THEN-ELSE block will then appear at the cursor. Now, just fill in the template.
- The VAX Language Sensitive Editor also provides online language-oriented help. For example, if you are editing a FORTRAN program and can't quite remember how to spell that OPEN-statement keyword, you can request online help for FORTRAN and the OPEN statement. You can then determine the valid options, and insert them into your program without ever leaving the editor.
- You can tailor the editor to your own environment. You can provide the definition of a language, a memo header, or other textual templates to the editor. You can also change or add definitions of the supported language and, therefore, conform to a specified programming convention.

®UNIX is a registered trade mark of AT&T Bell Laboratories.

- USING VAX DEBUGGER IN THE IMPLEMENTATION PHASE

The VAX Debugger is a powerful program development utility. It is designed to work in cooperation with the other program development productivity tools (VAX Language Sensitive Editor, VAX DEC/Test Manager, and many of the VAX languages).

The VMS program debugging facility (DEBUGGER) is a powerful and flexible tool that allows programmers to find errors in source code programs.

The DEBUGGER

- Is interactive. You can execute debugger commands from your terminal and see their effects immediately.
- Is symbolic. You can refer to program locations by the symbols you used for them in your program.
- Supports many languages. You use the debugger in the language of your source program. If your application is written in more than one language, you can change from one language to another in the course of a debugging session.
- Permits a variety of data forms and types for entry and display.
- Allows you to select and display your program's language statements.
- Has a screen mode that provides multiple windows for screen-oriented debugging.
- Has a debugger-defined keypad key definitions for your terminal's numeric keypad.
- Gives online help.

In DEBUGGER screen mode, you can divide a terminal screen into several different windows and specify what you want to see in each window. For example,

- Create a window into the source file you are debugging.
- Create a window to track the change in specific variables through the debug session.

Typically, one of those windows is a source display. When DEBUGGER encounters a break point, watch point, or exception condition, it displays the line of source code in which that event occurs (and several above and below it) in the source window.

DEBUGGER also allow you scroll through information on the screen. You can use the the terminal's arrow keys to move forward or backward through a window to review information that has appeared in that space.

You can invoke the VAX Language Sensitive Editor from the Debugger. When doing so, the cursor will be positioned on the current source code, unless you specify differently, in edit mode. After you have corrected the error to the source code, you can return to the DEBUGGER where you left off.

- USING DEC/CMS AND DEC/MMS TOGETHER TO MANAGE CHANGE DURING IMPLEMENTATION

  Programmers use DEC/CMS(Code Management System) and DEC/MMS (Module Management System) to manage the continually evolving software system in this phase. With code stored in CMS libraries, programmers now have a complete audit trail of every addition and change to the software system since the first source code file was included in the CMS library. Freezing baselevels and then recreating those modules, while the code is still evolving, is essential to reaching goals within time and budget.

  Programmers use DEC/Module Management System (MMS) to build and perform consistency checks on the system. DEC/MMS works in conjunction with DEC/CMS (Code Management System). When building the system, DEC/MMS only builds the parts of the system changed since the last time it was built. MMS, because it is integrated with the VMS operating system itself, builds your system reliably using a minimum of CPU time. Keeping the MMS description file in a CMS library allows it to evolve with the system too.

  DEC/MMS, used with DEC/CMS, keeps all programmers working on the same system without "version skew". Multiple versions of the software system can be enhanced at the same time and are available to all programmers. Since it is so easy to rebuild your system (all you do is use a single MMS command) you can always have an up-to-date version of your system available.

- USING RELATED VAX SOFTWARE PRODUCTS IN IMPLEMENTATION PHASE

  DECnet-VAX software products allow you to connect multiple VAX systems together and, therefore, increase your productivity. Communication with projects on other machines is possible with the VMS MAIL and PHONE utilities. Files may be shared even though you are accessing them remotely, over a DECnet link . Testing software in different machine environments is made easier by the SET HOST command, which allows you to log in to a remote system over DECnet.

CONFIGURATION MANAGEMENT IN THE IMPLEMENTATION PHASE WITH VAX/VMS - Because the implementation of a real software system is much more complex than the example presented here, you can quickly see that managing this complexity in a "real" program development life cycle can quickly become a challenge.

Configuration Management of your system can't be automatically done on VMS, but with careful planning and a solid program development methodology, you can use VMS tools to manage your software system's configuration.

To do this, you will need to identify all of your "configuration items" and put them, or references to them, in a CMS library or libraries. MMS "description files", descriptions of your up-to-date system, will need to be written with care and then updated as development continues. You need to define where your tests are to be placed, where your documents are to be built, and then you can use DEC/MMS to manage the configuration of your system.

The Development Plan, described in the Design Phase, is your first important step to managing this complexity. If you have a viable Development Plan, then you can separate the implementation of specific features of your software system into discrete stages or baselevels.

By using baselevels, you can build each new level of your system without rebuilding those you have already finished. This allows you to build each level in increments, i.e. in discrete stages. You only have to manage the new features of the next baselevel while the VAX management tools are taking care of the rest of the system.

When a baselevel is reached, all the documentation and source modules, the MMS description file, the test cases, and the benchmark files are placed into a CMS "class" to identify precisely the content of each baselevel. The CMS "class" is then made read-only so that it cannot be changed inadvertently. The MMS description file identifies the relationships between the modules and can be set up so specific test cases are run when a module changes. The CMS history file provides an audit trail of changes to the system. This adds traceability to your development process.

USING DEC/MMS TO DEVELOP TIMELY PROGRAM EXAMPLES IN USER MANUALS - Your project may need to guarantee the accuracy of all program examples, output, or error messages included in the documentation set. What the users find in the documentation should be the same as the program example output or error messages they see on their terminals. You can eliminate the many hours spent revising such examples when programmers revise a program and forget to tell the writer, or the writer makes changes to an example to improve its readability or style.

Using DEC/MMS, we

- Extend the description file, making the tested system (a null file used only for its time of creation time-stamp) depend on all the system components. The actions to build the tested system are running DEC/Test Manager with all the examples, and updating the time stamp file.
- Make the finished book depend on all the text files of the book and on the examples produced by the testing. The actions to build the book include running DSR. The DSR source file has .REQUIRE statements to pull in the examples (i.e. test files).

You may need to process the test results before they can be put in the book. For instance, you may want to enclose the example between .LITERAL and .END LITERAL statements so DSR will not try to change the format. Or you might want to discard a portion of the test output keeping only enough to demonstrate one point. That is easy too. Just write a small program in whatever language is convenient, and add a step to the description file which invokes the program. You might need to write several such small programs, but the effort will be repaid by the timeyou have made two or three drafts of each book.

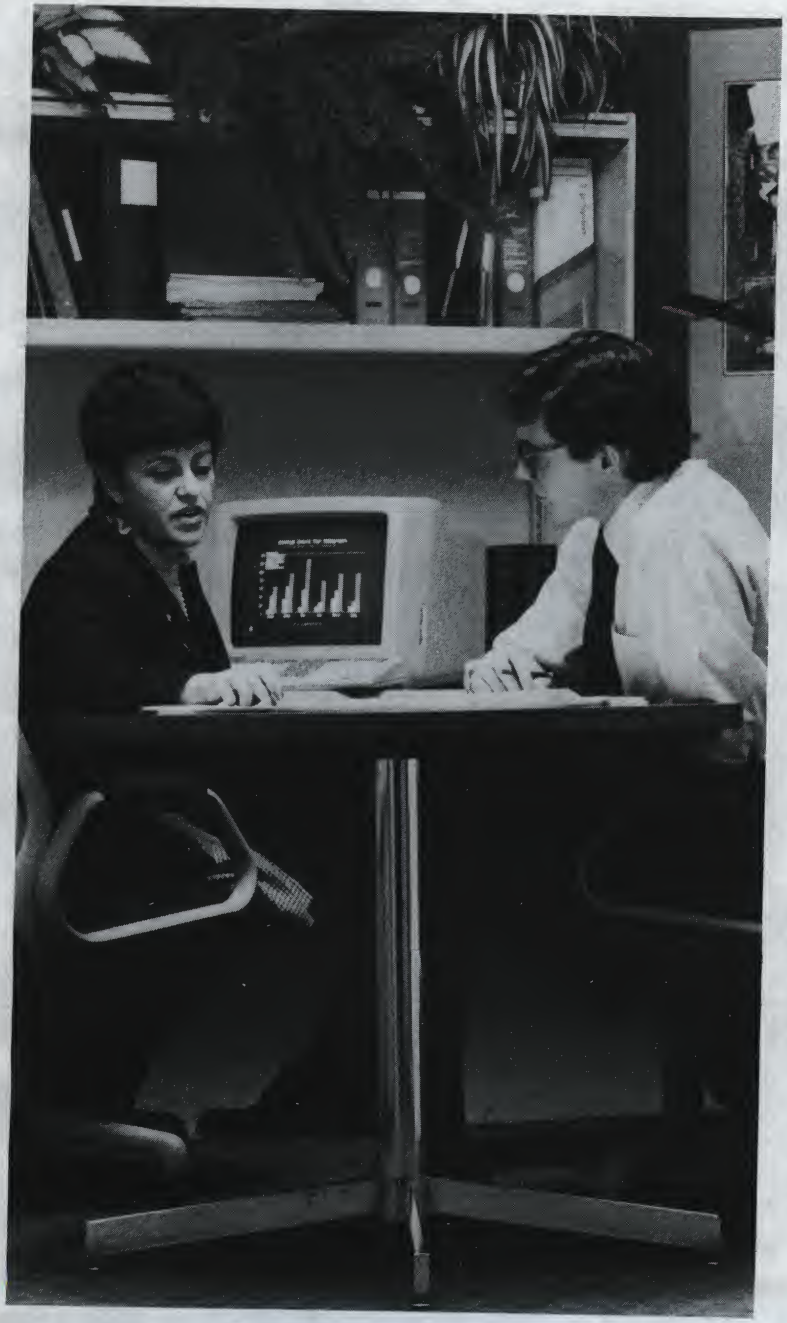
You can use this technique even if you do not have DEC/Test Manager. Simply invoke your program with redirected input and output You should check that the results are reasonable, either by looking at them, or by having additional programs validate the output.

Testing and Verifying the Software System with VAX/VMS

In order for the implementation phase to be completed, systemwide testing, debugging, and in-house testing of the software must be completed. The software system is considered a releasable product only when you are assured that it has been thoroughly tested and works well for users under a variety of conditions. Tests and system-builds should be integrated to make you as productive as possible in this phase. Figure 7-7 helps you to understand this.

Your programming staff's assignments are listed in the phase assignment row. Programmers 1 and 2 will continue to test and debug the software system. Programmers 3 and 4 have been assigned the task of building additional tests for the system. Many new problem areas in the system have been identified and now require testing. Programmers 5 and 6 will continue to test and write text for the online help facility. The documentation specialist must incorporate review comments into various manuals and prepare the documentation set for Field Test release. Field Test is the business of Phase 3; the intent is to make sure that a software product which has been thoroughly tested in-house can perform successfully in actual customer environments. If the software performs correctly, and the documentation is both accurate and helpful, then the product becomes a candidate for volume reproduction and sale to the general customer base.

Elements generated in Phase 2 are now online for reference or further development (column 2).



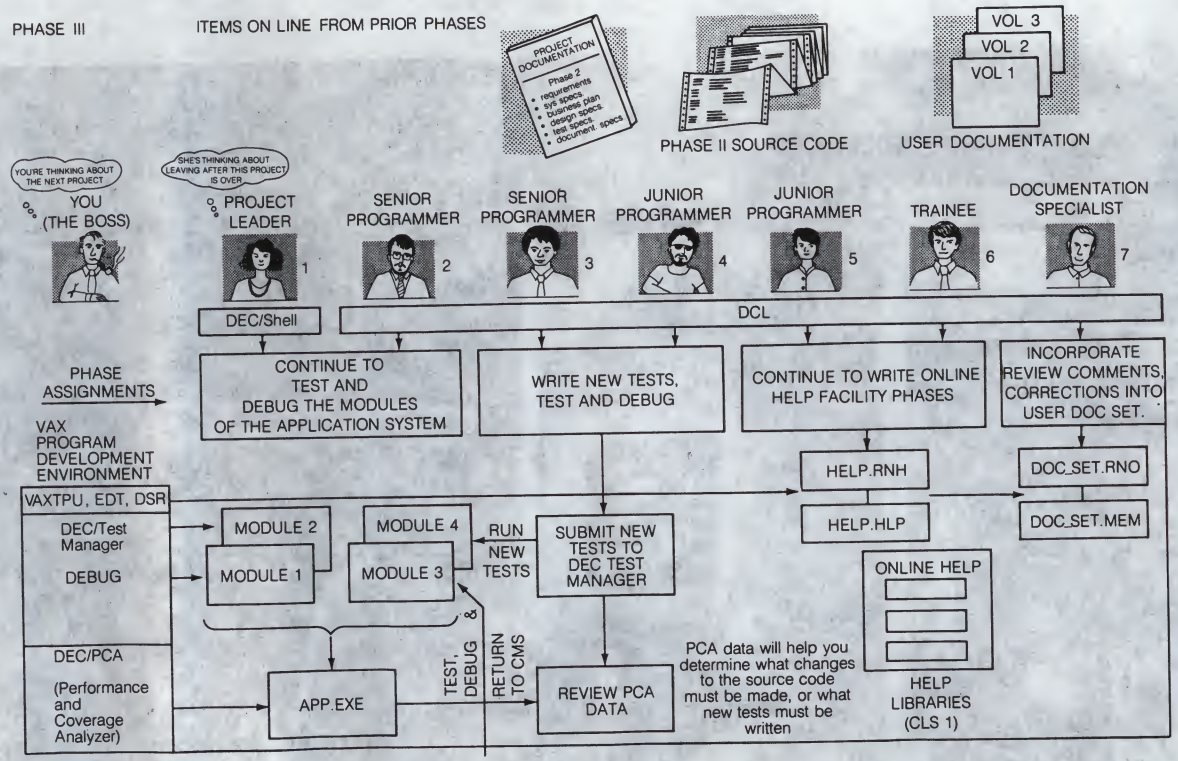


Figure 7-7 ■ Phase 3 — Testing and Verifying the Software System with VAX/VMS

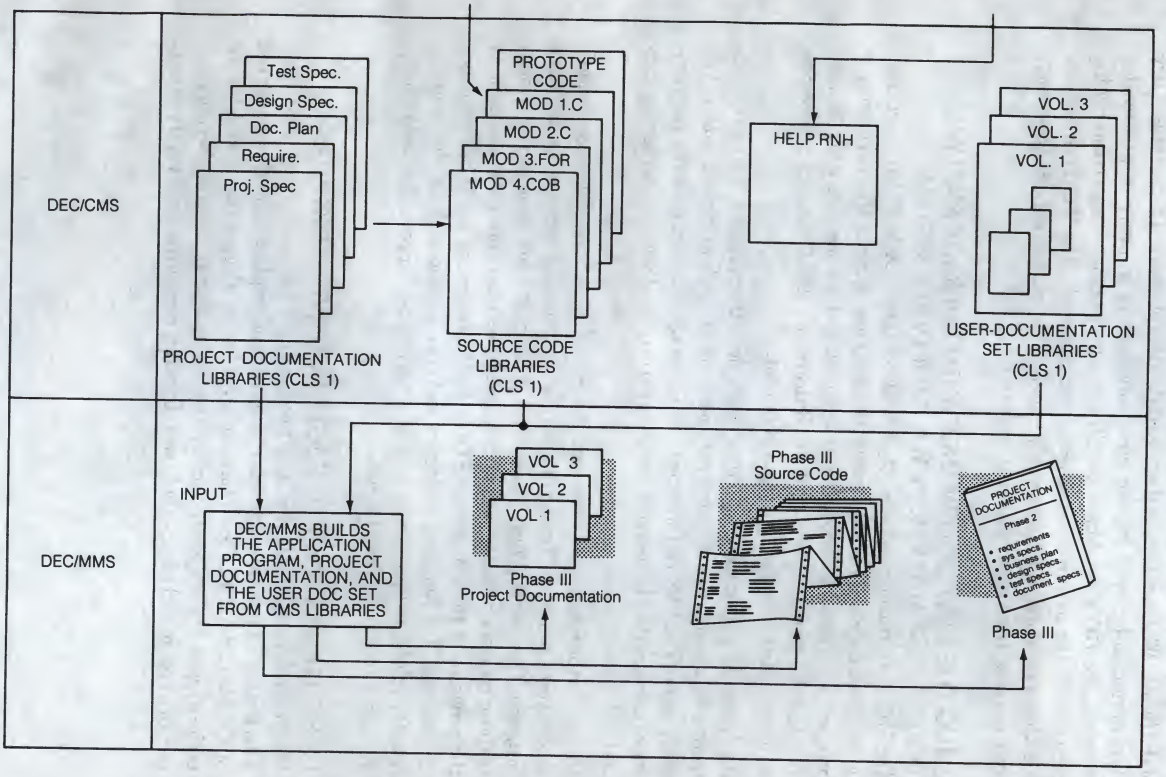


Figure 7-7 (Cont) ▪ Phase 3 — Testing and Verifying the Software System with VAX/VMS

The VAX/VMS tools used in this phase are listed, vertically, along the left side of the figure. We have already discussed the uses of many of these tools. They all continue to be used for the same functions as in previous phases (for example, the DEC/Test Manager still is used to manage regression-testing even though the nature of those tests have changed). The VAX Performance and Coverage Analyzer is extremely beneficial in analyzing how extensively your systems' tests cover your entire system.

- USING THE VAX PERFORMANCE AND COVERAGE ANALYZER (PCA) TO TEST AND VERIFY YOUR SOFTWARE SYSTEM

  The VAX Performance and Coverage Analyzer uses VAX/VMS AST (asynchronous system traps) services to sample the execution of events that occur in your program at runtime. It then writes data regarding those events to a file. You can then use the analyzer portion of the Performance and Coverage Analyzer to format the data into different types of output.

  Analyzing the performance of a program is done interactively. You can experiment with a number of different parameters until you find the information you really want. The VAX Performance and Coverage Analyzer provides performance coverage (PC) histograms, information on page-faults, system service calls and I/O calls.

  In the Performance and Coverage Analyzer's (PCA) test-coverage analysis mode, you can ask for breakpoints on every routine, statement, or path in a program. Then, after program execution, a command will show you how many of those routines or statements were actually executed during the execution of the program. A project using the DEC/Test Manager can put PCA commands into its DEC/Test Manager prologues and get coverage analysis while running tests for validity of the product. The user can also ask for coverage while reviewing tests in the DEC/Test Manager.

- USING THE DEC/TEST MANAGER IN THE TESTING PHASE

  When finishing a specific module, programmers will need to test their code. This testing is done over the entire implementation phase (and during the rest of the life cycle). DEC/Test Manager provides for a single test system for the entire project. It gives you the ability to select a subset of tests for checking only part of a system and also can help you efficiently review the results of those tests.

DEC/Test Manager

- Runs a collection of tests on your software system.
- Compares the output of each test it runs to output that are known to be correct.
- Provides a set of commands that automatically helps your project members execute a set of tests.
- Helps you quickly look at the results of those tests.

Many times, when testing for a certain condition or feature, your programmers may want to run one specific test with a combination of tests. With the DEC/Test Manager, it is simple to collect those specific tests for a run, run them, and review their results.

You can keep your test cases and benchmark files in a CMS library so that those tests can evolve with the system too. When you make a change to a system, you add a test case(s) to test the change. You also re-run the other test case(s) in your test system to verify the change has not "broken" anything else.

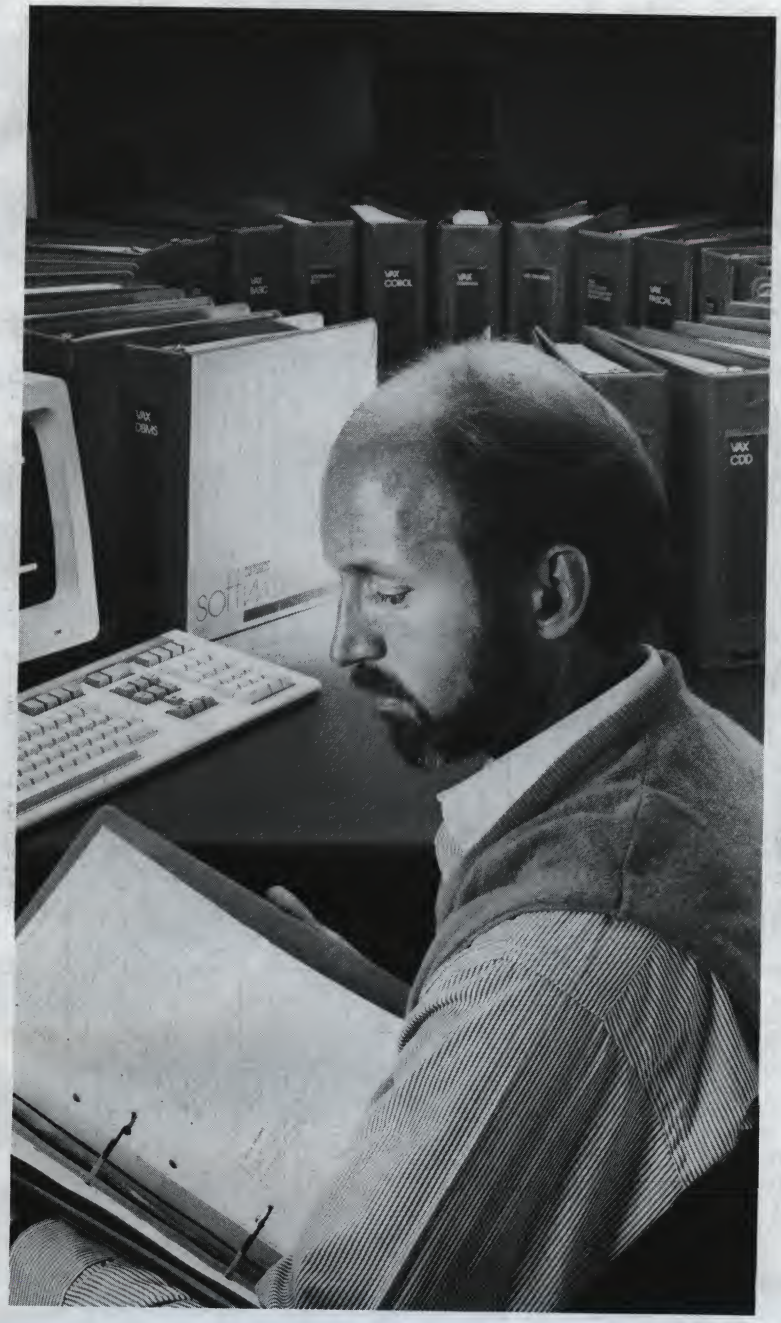
When a bug is fixed, you create a test-case to demonstrate the bug and capture the correct output Running these tests at a later date ensures that the bug stays fixed.

Maintaining the Software System With VAX/VMS

From many programmer's point of view, the program development life cycle is complete once the first four phases of the software system life cycle are completed. But you know much of the work (as much as 45 percent of it) on the software system doesn't even start until you enter Phase 4–volume production, sale to customers, and maintenance of the product in the field.

Quite often, once a large project is completed, programmers who have done most of the original design and coding work are put on other projects or seek new career opportunities. Replacing key personnel is never an easy task. When key programmers leave, they take a piece of your department. Therefore, to maintain your department's high productivity level, you must maintain continuity in the skill level of all programmers. Doing this insures that whomever you hire to replace these programmers is not going to spend a year getting "up to speed" on the system and learning the software system. The DEC/MMS description files, stored in a CMS Library, contain the information or how to build (or rebuild) the system as it currently exists, and how it existed at different points in the project's history.

By using the VAX/VMS products outlined in this handbook, all your code, documents, tests, and test results have been stored in CMS libraries for easy retrieval and reference. A new team member can easily use these libraries to determine the "who, what, when and why" of all work done for the first release of the software system.


VAX
BASIC
VAX
COBOL
VAX
PASCAL
VAX
DBMS
VAX
CDD

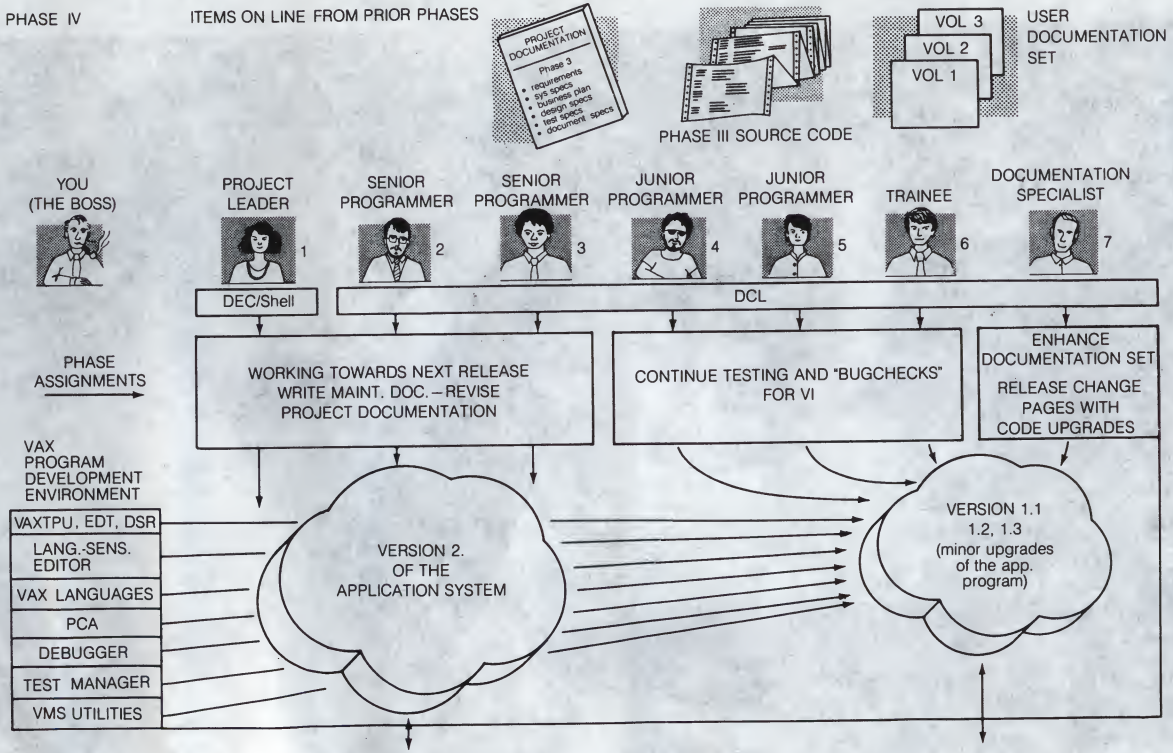


Figure 7-8 ▪ Phase 4 — Maintaining the Software System With VAX/VMS

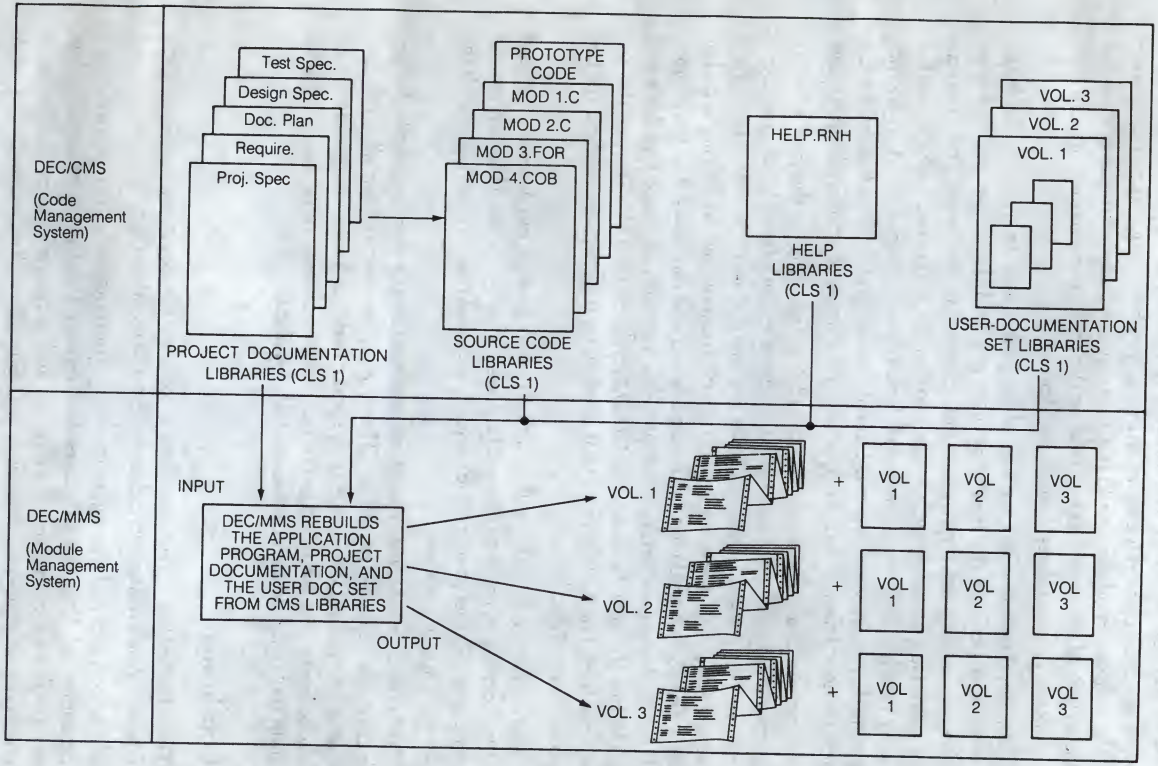


Figure 7-8 (Cont.) ▪ Phase 4 — Maintaining the Software System With VAX/VMS

Let's take a brief look (Figure 7-8) at your development department in Phase 4 of the software life cycle. The software system is now six-months old and in use at several hundred locations. Already, you have requests to change one of the software modules to add functionality not included in the first release due to lack of time you had to spend on the project, cost constraints, or lack of support from a related software system.

Often, development cycles are kept short deliberately. In this case, the initial release will perform the basic functions required. It is expected that after release, customers will request functions that were not included. These features are candidates for subsequent releases so that the system can evolve towards a closer match with the customer's needs than would be possible if an attempt was made to put everything in the initial release. This leads to Phase 0 for the next version of the product, if it appears economically or strategically justifiable.

You can see that programmers 1, 2, and 3 are working on the next software system. We should mention here that you have recently brought programmer 4 into the department. Although he is not familiar with the software or your department, he is able to be productive. He has a full history and audit trail of the project's code and internal documentation and can match it to the project. The VAX Language-Sensitive Editor helps him with language templates and walks him through the code. The DEC/MMS description file describes the system for him. The tools have made it possible for him to pick up where the last programmer 4 had left off.

Programmers 4, 5, and 6 will continue to make minor bug-fixes and general housekeeping on the first release. The documentation specialist has to revise the documentation set in much the same manner and issue change pages with the software upgrade releases.

This software system may evolve over the next five or six years, maybe longer, before it is replaced or retired. Therefore, it is essential that all records of changes to project documentation, source code, and user documentation be tracked over that entire period of time. Personnel may come and go, but an audit trail of the whole evolution must be available.

▪ Conclusion

This chapter has shown you how a software development project can use Digital's VMS Productivity Environment to your advantage.

The importance of developing software from an engineering perspective cannot be minimized. Our VAX languages, VAX tools and and VMS program development utilities are designed to make you and your development staff software engineers. Our products will allow you to build a software system in a single integrated environment with the industry's best engineered products.

Notes

Notes

Notes

Notes

VAX/VMS Software

Language and Tools Handbook

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our handbooks.

What is your general reaction to this handbook? (format, accuracy, completeness, organization, etc.)

_____

_____

_____

What features are most useful? _____

_____

_____

Does the publication satisfy your needs? _____

_____

_____

What errors have you found? _____

_____

_____

_____

Additional comments _____

_____

_____

Name _____

Title _____

Company _____ Dept. _____

Address _____

City _____ State _____ Zip _____

(staple here)

(staple here)

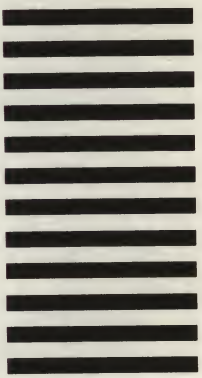
(please fold here)

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD, MASS.

POSTAGE WILL BE PAID BY ADDRESSEE



Digital Equipment Corporation
Corporate Communications Group
CFO 1-2/M92
200 Baker Avenue
West Concord, MA 01742

digital